USENIX LARGE INSTALLATION SYSTEMS ADMINISTRATION V CONFERENCE AUTUMN 1991

CONFERENCE PROCEEDINGS

Large Installation Systems Administration V

September 30 - October 3, 1991
San Diego, California

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: 510-528-8649

The price is $20 for members and $23 for nonmembers.

Outside the U.S.A. and Canada, please add
$11 per copy for postage (via air printed matter).

Past USENIX Large Installation Systems Administration Workshop and Conference Proceedings (price: member/nonmember)

Large Installation Systems Admin. I Workshop	1987	Phildelphia, PA	$4/$4
Large Installation Systems Admin. II Workshop	1988	Monterey, CA	$8/$8
Large Installation Systems Admin. III Workshop	1989	Austin, TX	$13/$13
Large Installation Systems Admin. IV Conference	1990	Colorado Springs, CO	$15/$18

Outside the U.S.A. and Canada, please add $8
per copy for postage (via air printed matter).

USENIX Association

Proceedings of the Fifth Large Installation Systems Administration Conference

September 30 – October 3, 1991
San Diego, California, USA

TABLE OF CONTENTS

Preface vii

Program Committee viii

Author Index ix

PLENARY SESSION

Tuesday (9:00-10:00) Chair: Elizabeth Zwicky

Opening Remarks and Announcements
Elizabeth Zwicky, SRI International

Keynote Address: Standards Efforts for System Administration
Martin Kirk, X/Open co. Ltd.

Similar But Different

Tuesday (10:30-12:00) Chair: Paul Moriarty

Managing Program Binaries In a Heterogeneous UNIX Network 1
Paul Anderson, University of Edinburgh

If You've Seen One UNIX, You've Seen Them All 11
Bob Arnold, ASK/Ingres Product Division

Software Maintenance in a Campus Environment: The Xhier Approach 21
John Sellens, Math Faculty Computing Facility, University of Waterloo

Integrating UNIX Within a Microcomputer-Oriented Development Environment 29
Peter Bumbulis, Donald Cowan, Eric Giguère, Terry Stepien, University of Waterloo

Panel: What Is System Administration

Tuesday (1:30-2:30) Chair: Rob Kolstad

Mail and Resource Accounting

Tuesday (3:00-4:30) — Chair: Steve Romig

The Design and Implementation of a Multihub Electronic Mail Environment 37
Nichlos H. Cuccia, Computer Sciences Corporation

A sendmail.cf Scheme for a Large Network 45
Tina M. Darmohray, Lawrence Livermore National Laboratory

SHARE II - A User Administration and Resource Control System for UNIX 51
Andrew Bettison, Andrew Gollan, Chris Maltby, Neil Russell, Softway Pty Ltd

System Resource Accounting on UNIX Systems 61
John Simonson, University of Rochester Computing Center

Backups, I

Wednesday (9:00-10:30) — Chair: Pat Wilson

A Next Step in Backup and Restore Technology 73
Rob Kolstad, SunSoft, Inc.

Issues in On-line Backup 81
Steve Shumway, SunSoft, Inc.

A Database for UNIX Backup 89
Jim Engquist, SunSoft, Inc.

A Distributed Operator Interaction System 97
Steve Shumway, SunSoft, Inc.

Panel: The IEEE Posix 1003.7 Sys. Admin. Standard

Wednesday (11:00-12:00) — Chair: Bjorn Satdeva

Tools

Wednesday (1:30-3:00) — Chair: Steve Simmons

A Flexible File System Cleanup Utility 105
J Greely, The Ohio State University

Fdist: A Domain Based File Distribution System for a Heterogeneous Environment 109
Bjorn Satdeva, /sys/admin, inc.; Paul M. Moriarty, MIPS Computer Systems, Inc.

Link Globally, Act Locally: A Centrally Maintained Database of Symlinks 127
Arch Mott, MIPS Computer Systems, Inc.

Watson Share Scheduler 129
Carla Moruzzi & Greg Rose, IBM T. J. Watson Research Center

Adding Additional Database Features to the Man System 135
Carl Shipley, Jet Propulsion Laboratory, California Institute of Technology

Dealing With Users

Wednesday (3:30-5:00) — Chair: Steve Romig

Modules: Providing a Flexible User Environment 141
John L. Furlani, Sun Microsystems, Inc.

Configurable User Documentation -or- How I Came to Write a Language with a Future Conditional 153
Mark A. Verber, The Ohio State University; Elizabeth D. Zwicky, SRI International

We Have Met the Enemy, An Informal Survey of Policy Practices in the Internetworked Community 159
Bud Howell, MTEK International, Inc.; Bjorn Satdeva, /sys/admin, inc.

Enhancing Your Apparent Psychic Abilities Through Software 171
Elizabeth D. Zwicky, SRI International

Backups, II

Thursday (11:00-12:00) — Chair: Bjorn Satdeva

Engineering a Commercial Backup Program 173
Jeff Polk & Rob Kolstad, SunSoft, Inc.

Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not 181
Elizabeth D. Zwicky, SRI International

Backups Without Tapes 191
Liza Y. Weissler, The RAND Corporation

Keeping Track

Thursday (1:30-3:00) Chair: Steve Simmons

Configuration Control and Management 195
Ed Arnold & Craig Ruff, National Center for Atmospheric Research

hobgoblin: A File and Directory Auditor 199
Kenneth Rich & Scott Leadley, University of Rochester

Monitoring Activity on a Large Unix Network with perl and Syslogd 209
Carl Shipley & Chingyow Wang, Jet Propulsion Laboratory, California Institute of Technology

SCRAPE (System Configuration, Resource And Process Exception) Monitor 217
Richard W. Kint, College of Engineering, University of Washington

Miscellaneous

Thursday (3:30-5:00) Chair: Pat Wilson

Packet Filtering in an IP Router 227
Bruce Corbridge, Robert Henig, Charles Slater, Telebit Corporation

Cloning Customized Hosts (or Customizing Cloned Hosts) 233
George M. Jones & Steven M. Romig, The Ohio State University

Staying Small in a Large Installation: Autonomy and Reliability (And a Cute Hack) 243
Edward Wang, University of California, Berkeley

Some Useful Changes for Boot RC Files 245
Steven M. Romig, The Ohio State University

Host Aliases and Symbolic Links -or- How to Hide the Servers' Real Name 249
John F. Detke, Octel Communications Corporation

Redundant Printer Configuration 253
Steven C. Simmons, Industrial Technology Institute

PREFACE

In the past five years, LISA has grown from being a small workshop to being a good-sized conference. The first USENIX workshop on Large Installation System Administration had a proceedings that was 46 pages long, a fifth the size of the current one. In the San Francisco bay area, the monthly meetings of the Bay-LISA group often reach the size of that original LISA workshop.

Why all this interest? As the average size of installations has moved inexorably up, the size threshold at which difficulties begin has stayed equally stubbornly exactly where it was to begin with. This makes large installation systems administration a topic of interest to more and more sites. Organizations that have had large numbers of computers for some time are also beginning to discover that networks and backups are both desirable, and that they require real work on the part of people who know what they're doing.

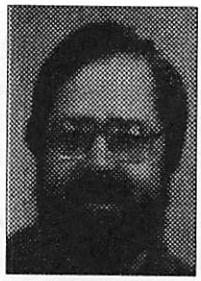
With the growth in the conference has come change. For the first time, there are three days of technical sessions; there is an alternate track; there is a vendor display. To our surprise, adding an extra day did not make the conference come out less densely packed. In fact, although we accepted more papers than ever before - nearly twice as many as last year - we also rejected more papers than ever before, making some very hard decisions in the process.

Many people have contributed to the making of this conference. I'd particularly like to thank the program committee (Paul Moriarty, Steve Romig, Bjorn Satdeva, Steve Simmons, & Pat Wilson); Judy DesHarnais; Lee Damon; Dan Klein; Rob Kolstad; and Ellie Young. In addition, I'd like to thank the many people who spent time and energy submitting papers and proposals.

Elizabeth Zwicky
Program Chair

PROGRAM COMMITTEE

Paul Moriarty
MIPS Computer Systems, Inc.



Steve Simmons
Industrial Technology Institute

Steve Romig
The Ohio State University

Pat Wilson
Dartmouth College

Bjorn Satdeva
/sys/admin, inc.

Elizabeth D. Zwicky
SRI International

CONFERENCE ORGANIZERS

Elizabeth D. Zwicky, *Technical Program Chair*
(SRI International)

Judith F. DesHarnais, *Meeting Planner*
(USENIX Association)

Dan Klein, *Tutorial Coordinator*
(USENIX Association)

Rob Kolstad, *Proceedings Formatting*
(SunSoft, Inc.)

AUTHOR INDEX

Paul Anderson	1
Bob Arnold	11
Ed Arnold	195
Andrew Bettison	51
Peter Bumbulis	29
Bruce Corbridge	227
Donald Cowan	29
Nichlos H. Cuccia	37
Tina M. Darmohray	45
John F. Detke	249
Jim Engquist	89
John L. Furlani	141
Eric Giguère	29
Andrew Gollan	51
J Greely	105
Robert Henig	227
Bud Howell	159
George M. Jones	233
Richard W. Kint	217
Rob Kolstad	73
Rob Kolstad	173
Chris Maltby	51
Paul M. Moriarty	109
Carla Moruzzi	129
Arch Mott	127
Jeff Polk	173
Steven M. Romig	233
Steven M. Romig	245
Greg Rose	129
Craig Ruff	195
Neil Russell	51
Bjorn Satdeva	109
Bjorn Satdeva	159
John Sellens	21
Carl Shipley	135
Carl Shipley	209
Steve Shumway	81
Steve Shumway	97
Steven C. Simmons	253
John Simonson	61
Charles Slater	227
Terry Stepien	29
Mark A. Verber	153
Chingyow Wang	209
Edward Wang	243
Liza Y. Weissler	191
Elizabeth D. Zwicky	153
Elizabeth D. Zwicky	171
Elizabeth D. Zwicky	181

Managing Program Binaries In a Heterogeneous UNIX Network

Paul Anderson - University of Edinburgh

ABSTRACT

This paper presents some of the techniques adopted in the Computer Science Department at the University of Edinburgh for providing a consistent user environment across a large network of heterogeneous workstations. These include system management techniques that allow non-privileged users to maintain and install network-wide application packages, as well as software and techniques for automatically distributing and replicating program binaries across the network.

Background

In a highly distributed network, it is often desirable to provide a consistent user environment across all workstations, so that users may move freely between systems, and the network can evolve with a minimum of disruption to the service. The extreme approach is illustrated by Project Athena at MIT [1] where an identical operating system, including the kernel and a large quantity of local software, is used on all workstations. Whilst this provides a totally uniform environment and good control over the available facilities, it is not suitable for many sites because of the difficulty of supporting this amount of software, especially where many different architectures are involved. This is a particular problem where there is a need to regularly adopt new and different hardware for technical, financial, or political reasons.

The system employed in the Computer Science Department at Edinburgh University[2], involves a compromise whereby a minimal base operating system, supplied by the hardware vendor, is overlaid with a standard local environment providing the higher-level facilities such as the shell, the window system, the editor, and other applications. [1] Clearly, this does not provide an absolutely identical environment across all platforms, but it does allow new machines to be incorporated quickly and the integration subsequently improved gradually by porting more of the standard environment, as necessary. The manufacturer's system software and specific enhancements also remain available (although unsupported) for those who wish to use them.

The other essential component, in providing the user with a consistent view of the network, is a virtual, network-wide filesystem. Home directories, for example, are physically located on a server in the user's home cluster, but are always referenced as **/home/user** and can be accessed from anywhere on the network. The manufacturer's implementation of NFS [3] together with the AMD automounter [4] provides a basis for such a virtual filesystem that is portable across many different platforms. The AMD maps are currently provided via NIS[5], but it is likely that these will be converted to Hesiod [6] in the future, allowing authority for a map to be delegated to the appropriate cluster. This use of standard NFS on a wide scale does incur several penalties, such as the need for a network-wide uid allocation scheme, and some difficult security issues which cannot be completely resolved without modifications to the NFS code itself. DNS [7] and NIS provide a global namespace for hosts and usernames.

Some filesystems, such as those containing home directories, are necessarily stored as a single live copy (since they need to be writable, and the traffic is relatively low). Other filesystems, however, such as the network-wide program binaries, need to be replicated across several servers, both to provide resilience against server failure and to distribute the load. The remainder of this paper presents some techniques that allow these *packages* to be maintained by users without superuser privileges, and system managers to control the distribution and amount of replication on a per-package basis. The basic aims are very similar to those of the *Depot* [8] framework, but a different filesystem organization, together with some local programs, provides a more flexible mechanism for controlling the distribution and replication of individual packages among servers.

Packages

Each *package* is allocated a user-id and all files belonging to that package are are created with the appropriate uid. The master distribution of the package is stored in the home directory, and any group of people can work on the package by changing their working uid to that of the package. A modified version of **su**, called **nsu**, allows users to change the working uid without supplying a password, providing that they are members of the the netgroup

[1] The GNU **bash** shell, MIT X11R4 and GNU **emacs**.

nsu_*package_name*. This allows systems managers to authorize users, simply by adding them to the appropriate netgroup in the NIS netgroup map. Direct logins with package uids are disabled in the password file, so that the "real" user is always identifiable. (The conventional approach of using Unix groups for this purpose was rejected for several reasons, including limits on the number of groups to which a user may belong, the need to make all package files group-writable, and differences in group semantics between different systems.)

Compiling packages for multiple architectures is often a problem because the object files created by a compilation for one architecture may interfere with those created on another architecture. Unless the package provides its own mechanism, this is usually handled by constructing *shadow trees* (a filesystem hierarchy identical to the master, but with each file replaced by a symbolic link to the master copy). The compilations are performed in the shadow directory, providing common source files, but separate object files for each architecture. The shadow trees are built with the utility **lfu** (below) and are stored on a separate filesystem (**/obj/local/***architecture*) which, since it contains only transient files, does not require backup.

On a standalone system, the final object files would be installed under **/usr/local** in subdirectories such as **bin**, **lib**, etc., similar to the usual hierarchy under **/usr**. Files that are common to all (or several) architectures, such as manual pages, or fonts, are stored under **/usr/local/share**, possibly with a symbolic link from other directories, in a similar way to **/usr/share** under SunOS. No files are installed directly into other directories under **/usr**, because of the problems involved in reinstating these files when the system is upgraded, or a new system is installed (a practically continuous activity in a large network).

The use of separate uids for each package has several additional benefits:

- It is easy to locate all the files corresponding to a certain package by running **find** with the required package name on the **/usr/local** filesystem.
- The system manager can obtain summaries of the space occupied by each package, using **du**, or the local program **lfck** (see Appendix I) which checks the filesystem for files with suspect owners, as well as providing a detailed disk usage summary.
- A regular daemon collects **README** files from the home directories of all the packages, appending them together into a single document that provides a summary of all the packages on the system. Users can then browse this document and locate the source (or at least the master distribution) for any package simply by looking in the home directory.
- Mail directed to a package account can easily be forwarded to the user(s) responsible for the maintenance of the package.

Distribution and Replication

The true network situation is more complex than the simple standalone model presented above, because servers need to supply binaries for more than one architecture, and multiple copies of the binaries need to be distributed among several servers. The general approach to this problem is to designate a *master* server for each package (usually in the home cluster of the user maintaining the package) which holds master copies of installed binaries for that package on all architectures. The *slave* servers then run a nightly job to update themselves from the various masters, and the clients mount **/usr/local** from a nearby slave carrying the appropriate architecture. Programs such as **rdist** [9] are designed to perform this type of update operation, but there are several problems which could not be solved adequately by existing software, and a local program **lfu** performs the server updates. Some of the important features include:

- The copying process should be as faithful as possible, including ownership and status of all types of filesystem object. For example, files with *holes* can be created by seeking past the end of the file; when these files are copied by most normal programs, the holes will be filled, usually generating a file larger than the source file.
- Given the large volume of software (currently over 1Gb for a single architecture and the common *shared* files), it is not generally possible for every slave to carry binaries for every package, so some mechanism is required to load easily configurable subsets of packages onto slave servers. However, to maintain a consistent view of the virtual filesystem, there must be some mechanism to ensure that files which are not resident on a particular slave are still accessible by the same pathnames.
- Slave servers will contain files from more than one master server, so it is essential that the set of files from one master server can be updated onto the slave without disturbing the set of files supplied from the other masters.
- Special actions are likely to be necessary when certain files are updated. For example, when replacing the binary for a daemon, it is essential that the existing binary is not immediately deleted, since it may be mapped into a running process. (It may however, be useful to automatically inform the system manager that the process needs restarting).
- A good log of all updated files is valuable, both for debugging, and to provide users with a list of files that have recently changed.

Since a network-wide filesystem is already supported, this can be used to access the master servers and no special network code is required in the update program.

One disadvantage of this nightly "bulk" updating of slave servers, is that the inconsistent state of the filesystem during the update could potentially cause problems for any programs running at the time. In practice, this has not proved to be a problem and any programs which are likely to be affected can be marked for special treatment (see the example for updating daemon programs below).

The Virtual Filesystem Hierarchy

Ordinary users are normally only concerned with the **/home** and **/usr/local** directories from the virtual filesystem. **/home** provides the home directories and **/usr/local** provides access to binaries for all the local packages. Package maintainers install packages in **/export/local** which is the master server for the the current cluster.

The master and slave servers for any particular cluster are also accessible as **/export/remote/***cluster* and **/usr/remote/***cluster*. This allows the update program to retrieve the latest version of a package from the appropriate master server. Files belonging to packages that are not carried on a particular slave server can be replaced by symbolic links to a slave server in a cluster which does carry the package. In this way, common packages can be carried by all servers, but packages that are normally of interest only to one particular cluster, can be carried on the slave servers from that cluster only. (although they are still accessible from everywhere else, under exactly the same pathnames, because of the symbolic links).

Figure 1 illustrates two clusters, each containing a master, a slave and one client:

- Package A is maintained on the master server in cluster A, and is copied onto the slave servers for both clusters.
- Package C is a specialist package for the users in the **vlsi** cluster. It is copied onto the **vlsi** slave server, but links are inserted into the local slave server so that the package is usable from the client of the local slave. Note that the actual value of the links will be **/usr/remote/vlsi/sun4/....**; the name **/disk/local** refers only to the mount point of the disk containing the binaries on the slave server (it is not

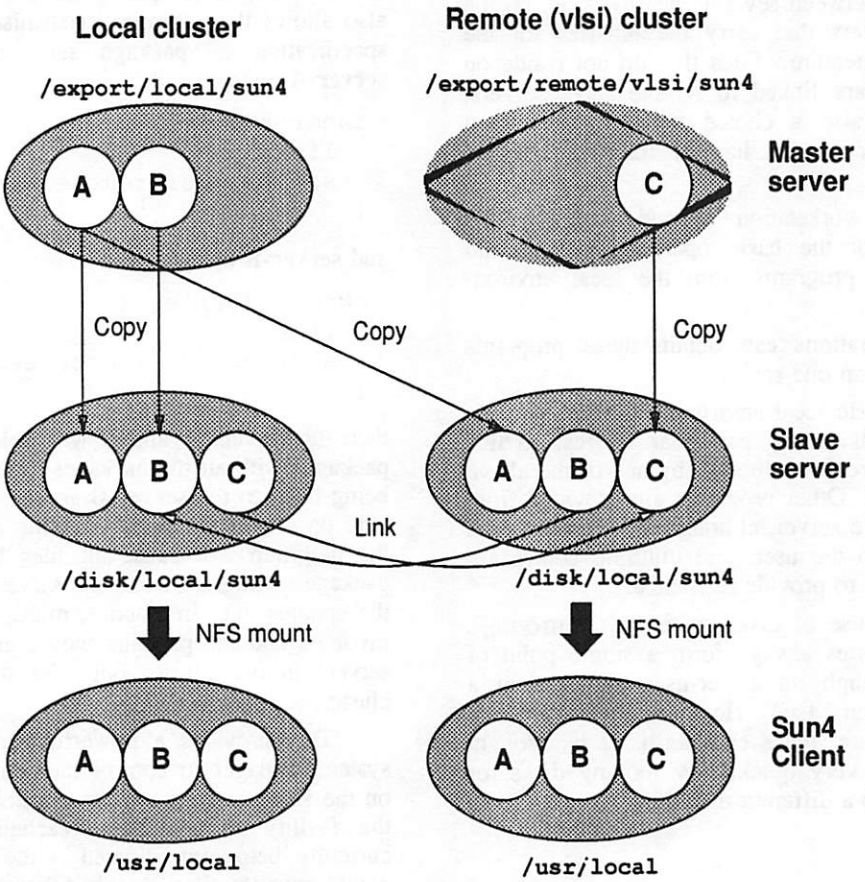


Figure 1: Updating servers

part of the network-wide filesystem).

- Package B is a similar specialist package for the local cluster. When this is referenced from remote slave, then the full name of the local cluster would need to be specified in the links.

Notice that all architectures are visible on the servers, but each client sees only its own architecture under **/usr/local**, and this view is functionally identical for every client.

Resilience

It is very important that the interdependence between machines is clearly defined and controlled; on a single machine, even links from public packages into users home directories, for example, could go unnoticed. In a distributed environment, this causes complex interdependencies and it is easy to reach a situation where a single workstation is dependent on too many servers and will fail when any one of the servers fails.

In most cases, the automounter is configured to mount **/usr/local** from the bootserver of the workstation, so that the workstation depends only on a single server for its basic operation. Where the workstation has no bootserver (or where the bootserver does not carry local binaries), the automounter will usually choose between several "nearby" (i.e., on the same wire) servers that carry the binaries for the appropriate architecture. Files that do not reside on the local slave are linked to remote slave servers, where there is also a choice between more than server. This arrangement has the following properties:

- A diskless workstation depends only on its bootserver for the basic operating system and most of the programs from the local environment.
- Other workstations can obtain these programs from more than one server.
- All of the basic local environment, and all applications which are of particular interest to this cluster, are provided directly by one of the above local servers. Other programs are provided from a remote slave server, although the remoteness is transparent to the user, and multiple copies are still available to provide resilience.
- Without the use of disk, or server, "mirroring", home directories always form a single point of failure (although on a per-user, rather than a per-workstation basis). However, in the event of a server failure, these can normally be brought online again very quickly by moving disks, or restoring onto a different machine.

The lfu Program

This program is responsible for copying files from a master server onto a slave server, implementing all of the features mentioned above, such as replacing certain files with symbolic links to other servers. In its basic mode of operation, **lfu** traverses the hierarchy of the master server in parallel with the hierarchy of the slave server. Files on the slave are deleted and/or copied from the master to make the slave into a faithful copy of the master. This basic operation can be modified by providing a *script* to **lfu** containing a list of *conditions* and *actions*. The *actions* are applied to any file matching the *conditions*, in place of the default action. For example:

```
owner=vlsitools {
  link;
  source=/usr/remote/vlsi;
}
```

will update all files on the slave server from the master server, except that files (or directories) owned by the package **vlsitools** will be replaced with links to the corresponding file on one of the servers in the **vlsi** cluster (automounted under **/usr/remote/vlsi**).

The use of ownership to identify the packages avoids the need to specify large explicit filesets. It also allows the netgroup mechanism to be used for specification of package sets. For example, if **server-A** updates with

```
owner=@bfiles {
  link;
  source=/usr/remote/server-B;
}
```

and **server-B** updates with

```
owner!=@bfiles {
  link;
  source=/usr/remote/server-A;
}
```

then the servers A and B will hold disjoint sets of packages with all the packages in the netgroup **bfiles** being held on the **server-B** and all other files being held on **server-A**. Simply adding a new package to the netgroup will cause all files belonging to that package to migrate from one server to another when the updates run. In practice, multiple servers will be involved and the package would migrate off all the servers in one cluster onto the servers in another cluster.

This provides a powerful mechanism for the system manager to control the usage of disk space on the slave servers and the possibility of extending the facility to provide a cacheing behaviour is currently being investigated - the following script would migrate files that had been accessed in the last two weeks onto the local server and replace less

frequently used files by a link to the copy on a larger server:

```
access > 2wks {
  link;
  source=/usr/remote/mainserver;
}
```

The examples below show some other features of the **lfu** program.

Updating Daemon Programs

If the file **daemons** contains the names of the daemon programs, then the script:

```
path=@daemons {
  exec restart $F;
  keep;
}
```

will execute the shell script **restart** (with the name of the updated program as a parameter) whenever one of these programs is updated[2]. This shell script could automatically restart the daemon or mail the system manager to intervene manually. The **keep** action indicates that the old version of the program on the slave server should be renamed, rather than being deleted, since the text may be mapped into a running program. (In this example the renamed version of the file would be deleted automatically next time the update runs, since the corresponding master file would not exist).

Programs That Must Be Owned By Root

Some programs need to be owned by root, perhaps because they need to run "setuid". If such files exist on the master server, they can be difficult to identify and can cause problems across NFS if the **root** user is mapped onto **nobody**. The following script will change the ownership and permissions of any files specified in **rootfiles**, as they are copied. Any files owned by root on the master will generate an error message and not be copied; this ensures that the **rootfiles** are the only root-owned files on the slave server, providing a useful security check.

```
path=@rootfiles {
  chown root; chmod u+s;
}
else owner=root {
  error "Root file on master";
}
```

Updating From Multiple Servers

When a slave server updates from more than one master server, the files supplied by one of the masters must not be deleted when updating from the other. For example, the following script could be used to update the local slave server of Figure 1 from the local master:

```
owner=package-C {
  ignore;
}
```

When updating from the remote **vlsi** master, the following script could be used:

```
owner=package-A |
owner=package-B {
  ignore;
}
else owner=package-C {
  link;
  source=/usr/remote/vlsi/sun4;
}
```

Netgroups are particularly useful here to define the sets of packages supplied by each of the master servers, and the sets of packages carried by each slave.

Some Difficulties

The following paragraphs illustrate some issues that have required special attention:

Package Installation

Care is needed when installing packages, since the binaries must be installed on the master server (**/export/local/**), but any references made by the running programs must be made to files on the current slave server (**/usr/local**). Most installation procedures are not designed to handle this situation, and it is easy to inadvertently install packages that make direct references to the master server when they are running. Such programs will continue to operate, but this introduces an unwanted dependency between the client and the master server. It may also introduce an excessive load on the master server. These dependencies are not always easy to detect; sometimes the automounter can be seen unexpectedly mounting a directory from the master, or, more usually, the dependency is only noticed when the master server is shut down.

Another difficulty for the package maintainer is the time delay between installing a package onto the master and having it propagate to the slave, where it can be tested. For complex installations, a particular workstation may be configured to reference certain binaries directly off the master server, so that the installation can be tested without waiting for changes to propagate to the slaves[3]. The state of a particular package can be "frozen" on all the slave servers by added the package name to a netgroup which is

[2]Unfortunately, the stateless nature of NFS normally makes it impossible to automatically detect files which are currently in use.

[3]Forcing updates is also possible, but this leaves slave servers in inconsistent states and can be rather slow.

ignored by the **lfu** scripts. This is useful when working on complicated installations or version updating, to prevent incomplete or inconsistent installations being propagated.

Conflicting filenames

With a large number of packages available, **/usr/local/bin**, for example, becomes very large and there is an increasing chance of the same filename being used by more than one package. Where a package (such as **X11**) has a large number of associated binaries, these are often moved to a subdirectory and arrangements made to include the subdirectory in the user's path, when appropriate. Normally, however, if two packages from different master servers include a file with the same name, the conflict may not be immediately apparent, although it should be possible to detect.

Updating Multiple Slaves

With a large number of slave servers, there can be a problem with too many slaves attempting access the same master at the same time. Currently, this is prevented by having some slave servers update from other slaves, rather than directly from the master. This idea could be extended to a hierarchy of slaves, which would prevent any one particular server becoming overloaded.

In a typical update run, 80-90% of the cpu time (and NFS traffic) generated by **lfu** is incurred in scanning the filesystems to locate files which have been changed. In the case of identical slave servers, it should be possible for just one of the servers to perform the scan, and pass on information about the required updates to the others. This is currently being investigated.

Conclusions

The techniques described above have evolved over the past three years on the network within the Computer Science Department, and have recently been extended to include clusters belonging to other small groups. Currently, 200-300 workstations are supported with three master servers and some tens of slave servers. Four major architectures are supported[4], and several others are included with a lesser degree of support.

In practice, the network is continually evolving and there are always some clusters and individual machines that are only partially incorporated. Certain clusters may decide (perhaps for licencing reasons) not to carry a particular package at all, or not to provide access to a particular group of home directories (perhaps for security reasons). The ability to support this degree of flexibility at the same time as providing a consistent and stable user environment has been one of the most important benefits.

The concept of providing a uniform environment across a heterogeneous network has undoubtedly been successful, and is popular with users. The need to attempt this without modifications to the hardware vendor's base operating system has lead to some obvious visible differences between different platforms and many difficulties that could have been avoided by running a completely standard system. However, a reasonable compromise has been reached and new hardware can usually be incorporated, with an acceptable degree of integration, very quickly.

The method adopted for management of software packages has generally been very successful, on the present scale. System managers are usually unaware of the detailed changes to individual packages, but are able to monitor and control the placement of the binaries very easily, whilst users are unaware of the underlying services and can use any software from any workstation.

The success of the current system is leading to its adoption by other clusters and, although we expect the basic concepts to scale reasonably well, the wider scale is expected to emphasize the difficulties of using standard available software, such as the vendor's implementations of NFS. As these kind of problems become more widespread, we hope that vendors will begin to incorporate solutions (such as the Kerberos [10] enhancements to NFS) into their own products.

Acknowledgements

The implementation and evolution of the network would not have been possible without the continual efforts of all the systems staff in the Computer Science Department. In particular, Alastair Scobie and Russ Green have been actively involved in the design of many of the concepts discussed above.

Author Information

Paul Anderson graduated in Pure Mathematics from the University of Wales in 1977. He taught Mathematics and Computer Science at the North East Wales Institute of Higher Education until 1984 when he became system manager for the Institute, establishing a new computer centre and software development team. In 1988 he moved to the University of Edinburgh as Systems Development Manager with the Laboratory for the Foundations of Computer Science, where he is currently managing the laboratory network and working with other system managers to improve the integration and administration of the university networks. Paul can be reached by mail at the Laboratory for the Foundations of Computer Science; Department of Computer

[4]Sun SPARC (SunOS), Sun 68000 (SunOS), HP9000 (HP/UX) and DECstation 5000 (Ultrix).

Science; University of Edinburgh; King's Buildings; Edinburgh; EH8 3JZ; U.K. Reach him electronically at paul@dcs.ed.ac.uk.

References

1. Jennifer G. Steiner and Daniel E. Geer, Network Services in the Athena Environment, Project Athena, Massachusetts Institute of Technology, Cambridge, MA 02139.
2. Paul Anderson, Installing Software on the Computer Science Department Network, Department of Computer Science, University of Edinburgh, Edinburgh, August 1991.
3. Sun Microsystems, "Network File System: Version 2 protocol specification," in *Network Programming Guide*, pp. 168-186, Sun Microsystems, 1990.
4. Jan-Simon Pendry, AMD - An Automounter, Department of Computing, Imperial College, London, May 1990.
5. Sun Microsystems, "The Network Information Service," in *System and Network Administration*, pp. 469-511, Sun Microsystems, 1990.
6. Stephen P. Dyer, The Hesiod name server, Project Athena, Massachusetts Institute of Technology, Cambridge, MA 02139.
7. Sun Microsystems, "Administering the Domain name service," in *System and Network Administration*, pp. 513-554, Sun Microsystems, 1990.
8. Kenneth Mannheimer, Barry A. Warsaw, Stephen N. Clark, and Walter Rowe, "The Depot: A Framework For Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *Proceedings of LISA IV Conference*, 1990.
9. Sun Microsystems, "rdist (1)," in *SunOS Reference Manual*, Sun Microsystems, 1990.
10. Jennifer G. Steiner, Clifford Newman, and Jeffrey Schiller, Kerberos: An Authentication Service for Open Network Systems, Project Athena, Massachusetts Institute of Technology, Cambridge, MA 02139.

Appendix I: lfck

The following example shows a fragment of output from the **lfck** program summarizing the disk usage (by package, and by architecture) on one of the master servers. The figures in brackets represent the space occupied by files exported to the slave server, and the main figures represent space occupied on the master server only (home directory of the package, and compilations under **/obj/local**). Notice that this is only a section of the real output, so the row and column totals do not correspond to the figures in the table.

Package	share	sun3	sun4	Total Mb
X11 Release 4	24.1 (24.7)	36.8 (29.9)	36.0 (30.2)	615.6
Poplog	139.7 (29.9)	25.2 (28.9)	0.1 (27.1)	297.1
GNU Emacs	185.9 (11.1)	0.1 (11.6)	0.1 (11.0)	251.5
TeX	85.3 (41.5)	22.5 (17.6)	24.4 (13.3)	233.1
InterViews	35.7 (1.7)	9.2 (6.4)	28.8 (18.7)	113.8
GNU C Compiler	30.7 (2.1)	10.3 (2.0)	11.4 (2.0)	110.5
Centaur	52.4	11.4 (14.5)	0.1 (18.7)	96.8
Modula 3	41.2 (0.1)	0.0	31.8 (12.7)	94.6
Generic graphics	7.7 (0.8)	17.0 (15.0)	11.3 (4.2)	89.1
IE Editor	0.1	0.0	0.0	0.1
Local Admin Data	0.1	0.0	0.0	0.1
Total Mb	2621.8 (235.2)	288.5 (318.7)	282.2 (370.5)	4742.7

lfck can also detect files which are not owned by a valid package and apply a number of heuristics to suggest the correct owner.

Appendix II: lfu

lfu currently accepts the following *conditions* and *actions*:

Conditions:

name=*value* Matches the name of the file against a regular expression. An explicit list of files can also be specified.

path=*value* Matches the pathname of the file against a regular expression. An explicit list of pathnames can also be specified.

owner=*value* Tests the owner of the file. A netgroup can also be specified.

group=*value* Tests the group of the file.

type=*value* Tests the type (mode) of the file.

age*[><=]value* Tests the age (mtime) of the file.

access*[><=]value* Tests the last access time of the file.

Actions:

update The default action - update the file if the source is more recent than the destination.

ignore Ignore the file.

delete Delete the file.

preserve Update, if necessary, but do not delete.

chmod *mode* Change the mode (perms) of the file.

chown *owner* Change the owner of the file.

chgrp *group* Change the group of the file.

link Link objects rather than copying.

shadow Copy directories, but link other objects.

source *value* Specify the source directory for links.

log Log any changes to this file.

logall Log any examination of this file.

error *msg* Report specified error message.

exec *command* Execute specified command whenever file is updated.

keep Rename file rather then deleting.

fill Do not attempt to duplicate "holes" in files.

newtime Do not duplicate the mtime when copying files.

force Update files regardless of the file times.

follow Follow symbolic links.

If You've Seen One UNIX, You've Seen Them All

Bob Arnold - ASK/Ingres Product Division

ABSTRACT

The title is absurd, of course. One of the major challenges in a very heterogeneous UNIX environment is to make system administration tools handle differences among UNIX variants and differing local configurations. In short, how can one write tools that work everywhere? This paper describes an approach to coping with this challenge (and at the same time making the title seem, well, at least a little bit less ridiculous). The heart of the technique is to use a Bourne shell script to research a local environment. The research results are written to a configuration file, which is itself a shell script. It is thus suitable for direct sourcing by any **sh** script, and can be easily consulted by other kinds of programs. This technique depends a) on the availability of a common tool set, which is available in the Version 7 utilities, and b) a dynamic tool design approach which understands and copes properly with the differences. Related techniques and experiences are also discussed.

Introduction

In their never ending search for better ways of doing things, vendors are fond of tailoring their version(s) of UNIX. The sheer variety of differences makes life difficult for a system administrator. Differences among UNIXs are commonly found in these areas:

- Procedures for changing configuration files
- Configuration file location/name
- Configuration file format
- Command syntaxes
- Program availability
- Program capabilities and behaviors
- Program outputs
- Program location/name

One of the keys to administering many systems is to write tools to handle as many common tasks as possible. This may make it possible to automate common tasks, spend less time on them, or even hand them off to other people. Yet in an environment with systems from many vendors, writing tools that work with all systems can be difficult.

Environment

The Data Center at the ASK/Ingres Products Division is responsible for supporting (at various levels) almost 300 UNIX systems from well over 20 vendors. Each vendor may be represented by one or more operating systems, sometimes running multiple major releases on several different CPU architectures. New hardware and operating systems arrive frequently, and must be integrated into our environment quickly. This process has been going on for years, so we also have a number of relatively old UNIX platforms to support.

Early Attempts

Under these circumstances, we need a strategy for making our tools work on all these machines. Early attempts at getting tools to work "everywhere" included variations on the theme below. Naturally they broke on various System V / BSD hybrids.

```
if test -f /etc/inittab ; then
    # make lots of System V
    # assumptions
else
    # make lots of BSD
    # assumptions
fi
```

We found that this kind of research code had to be repeated in many tools. And it was tedious to make improvements to the research code when the same fixes had to be inserted into many tools.

In a few tools we tried to maintain different versions for different platforms. This created small porting problems and large source code maintenance nightmares.

Configuration Research Strategy

Eventually, all of this became ridiculous. It was decided that we would try to dynamically test for everything we cared about, from where to find **csh** to **echo**'s no-newline syntax. We could build one script to do all the research and report the results in a flat file which could be consulted by other tools. As it happens, all our tools that have to run everywhere are Bourne shell scripts, so it quickly became obvious that the flat file should be written as a Bourne shell script too. Thus the flat file could be directly sourced by **sh** scripts, and without too much work could be consulted by other kinds of programs.

We decided to maintain only one current version of each tool. If a tool had to run on multiple platforms, then it had to be coded to handle OS differences. In other words, such a tool had to work as written, without porting. Tools learned about their environment from the research tool's report.

Implementation

Implementation began by creating **saenv.setup**, the configuration research tool. (The accepted pronunciation is unfortunately awkward: "ess-ay-env-dot-setup".) This script investigates topics of interest to our toolset. The report it writes is called **saenv** (said "ess-ay-env" or "ess-ay-ee-en-vee"), and it describes the local system administration environment. Any **sh** script can source it directly, using the Bourne shell "." command.

When a new master version of **saenv.setup** is created, it is distributed to each host and run. Each host thus has its own version of **saenv**. This file is static, until a new version of **saenv.setup** comes along.

To make use of **saenv**, we distribute another file, **sahead** ("ess-ay-head"). This script is sourced by every tool; in fact most tools source **sahead** before doing anything else. **sahead** (and the tool using it) exits with an appropriate error message if it finds any problems with **saenv**; otherwise it sources **saenv** in turn. **sahead** also sets variables which are constant throughout our environment. (It has one last duty, which is to figure out the name of the calling script.)

In this way, each tool quickly sets many variables that it needs. From **sahead**, it gets variables which are global to our environment. From **saenv**, it gets variables which are local to the host the tool is running on.

Each host administered by the Data Center has a setup that looks like Figure 1. A few comments are in order. First, **/etc/dist** is our target for distributed configuration files like **/etc/hosts**. Distributed system administration tools live in **/etc/dist/bin**, including the distribution script **upd-dist** ("update distribution") itself, and **realdir**. This last script is used by **saenv.setup** to determine the real location for certain files - for example, does **aliases** live in **/etc** or **/usr/lib**?

Although they are shell scripts, **saenv.setup**, **sahead**, and **saenv** have no execute permissions. The last two are intended only for sourcing by other tools. The preferred way to run **saenv.setup** is to explicitly use **sh**, because we want to make sure the script isn't run by some other interpreter (this is probably overkill, but no harm done).

```
sh /etc/dist/saenv.setup
```

This is how **upd-dist** does it whenever it uploads a new version of **saenv.setup**.

Similarly, because they are distributed but not executable, **saenv.setup** and **sahead** are kept in **/etc/dist**. Since **saenv** is not distributed, we put it in **/etc**.

Examples

After some initialization, the first thing that **saenv.setup** does is to determine a valid `PATH` (see Figure 2). It tests a list of desirable directory names to see if they exist, adding each directory it finds to a variable `$path`. When the search is done, **saenv.setup** does

```
PATH=$path; export PATH
```

for itself. It also puts the same code into the temporary version of **saenv** (see Figures 2 and 5). After initial comments, setting the `PATH` will be the first thing that **saenv** does for any calling script. The **saenv.setup** script installs the temporary version in **/etc** if all goes well. Note that **saenv** contains comments about the purpose of each variable.

The list of desirable directories is compiled based on our experience with different UNIXs. Many of them are common to System V, BSD, and their derivitives. Others are vendor specific.

The **cron** setup is a good place to illustrate a dynamic testing approach. We can show examples of several of the items listed in the introduction:

- Procedures for changing configuration files (do we have to run the **crontab** command to install a new **crontab** file)
- Configuration file location/name (the file **/usr/lib/crontab** vs. an alternate location **/usr/spool/cron/crontabs/root**)
- Configuration file format (4.3BSD and derivitives vs. everybody else)

```
drwxr-xr-x 3 root /etc/dist                  # for files we distribute
drwxr-xr-x 2 root /etc/dist/bin              # distributed tools go here
-rwxr-xr-x 1 root /etc/dist/bin/realdir      # used by saenv.setup
-rwxr-xr-x 1 root /etc/dist/bin/upd-dist     # distribution tool run nightly by cron
-r--r--r-- 1 root /etc/dist/saenv.setup      # research tool which creates saenv
-r--r--r-- 1 root /etc/dist/sahead           # sourced by every tool, sources saenv
-r--r--r-- 1 root /etc/saenv                 # host-specific sysadmin environment
```

Figure 1: Setup of system administration environment

```
## This is saenv.setup which should be 444 root.  [rest of header deleted]

## set PATH environment variable.
# 1) We always want the stuff in $local, need to test the rest.
# 2) Want BSD dirs before SysV dirs, so /usr/ucb before {/usr,}/bin.
# 3) Put OS bin directories, then vendor stuff, and local stuff last.
guesscommon="/usr/ucb /usr/bin /bin /etc /usr/etc"
guessvendor="/usr/sbin /usr/amdahl/bin /usr/lbin /sbin"
for dir in $guesscommon $guessvendor ; do
        if test -d $dir ; then
                path=$path:$dir
        fi
done
path=`echo $path | sed -e 's/^://'`      # strip leading colon
path="$path:$DTGTBIN:/usr/local/bin"     # add local directories
PATH="$path" ; export PATH               # PATH now set for saenv.setup

## initial setup of temporary saenv
tsaenv=/tmp/saenvt$$     # temporary sysadmin environment file
echo "$HEADER" > $tsaenv          # create $tsaenv
echo "PATH=$PATH          # exportable PATH" >> $tsaenv
echo "export PATH" >> $tsaenv
```

Figure 2: Sample code from **saenv.setup** - PATH

```
## CRONCMD_B - do we use the crontab command?

bool=false
if ( crontab -l ) > /dev/null 2>&1 ; then
        bool=true
fi
echo "CRONCMD_B=$bool    # do we use the crontab command?" >> $tsaenv

## CRON_P - Where does the (root) crontab file really live
x=`/etc/dist/bin/realdir crontab /usr/etc /etc /usr/lib /private.MC68020/usr/lib`
if test -z "$x" ; then
        if [ -d /usr/spool/cron/crontabs ] ; then
                cpath=/usr/spool/cron/crontabs/root
        fi
else
        cpath=$x/crontab
fi
echo "CRON_P=$cpath      # path to real crontab file" >> $tsaenv

## CRON43_B - Are we a 4.3 crontab host?
# count lines which aren't comments and have "root" as the sixth field
x=`awk '$1 !~ /^\#/ && $6 == "root" {print}' $CRON_P | wc -l`
bool=false
if test $x -gt 0 ; then
        bool=true
fi
echo "CRON43_B=$bool     # is the crontab file in 4.3BSD format?" >> $tsaenv
```

Figure 3: Sample code from **saenv.setup - cron** variables

The code to research these items is shown in Figure 3, and the results can be seen in Figure 5. Since **saenv.setup** has already determined a valid PATH, the simplest way to see if the **crontab** command is on this system is to run a benign `crontab -l` command. This is done in a subshell so we can throw away any `crontab: not found` error messages from **sh** (if it is not on the system) or **crontab** output (if it is). The exit status of the **crontab** subshell tells us what we need to know, and we set `CRONCMD_B` accordingly.

The full path to the **crontab** file, including the directory it really lives in, is established in the `CRON_P` paragraph. (Our tools do not need to handle **crontab** files for non-root users.) The **realdir** script is given a filename to look for, and a list of guesses about where to find it. The list of guesses is again based on our experience. The `CRON43_B` paragraph assumes that if we find the word "root" in the sixth field in the **crontab** file, then it is in 4.3BSD format.

This brings up the problem of robustness. As it is currently written, the test for whether or not a system's **crontab** file is in 4.3BSD format could fail. When we install a master **crontab** file, local entries at the beginning of the file are preserved. In our environment, the primary users of a machine have root access to that machine. Although unlikely, a superuser of a non-4.3BSD host could install a `root` command as a local **crontab** entry. This would cause the test to give a wrong answer. We would need to ask the user to change the program name, or develop a more robust test.

Like system administrators everywhere, we always have too much work to do. So, we get a tool working and leave it alone. If it breaks frequently enough or in a really ugly fashion, we make it more robust and repeat the cycle.

When creating these tests, you can end up learning obscure details of a program's behavior. If you are looking for the BSD **dump** backup program, you have to make sure you don't find the System V **dump** filesystem description tool. One way to do this is to test known paths to BSD `dump` with the `W` flag. The heart of the research code is shown in Figure 4, and the results are in Figure 5.

The first `if ls ...` is an attempt to emulate `if test -x ...` ; the <space><star> in the **grep** regular expression is required to handle a bug in one vendor's `ls -dF` which puts a trailing space after the asterisk. The second `if` is the main test. It relies on two conditions. First, it expects BSD

```
## DUMP_P - Find path to BSD dump backup program - avoid SysV dump
.....
for guess in /usr/etc/dump /bin/dump /etc/dump /etc/bsddump \
/etc/fdump /etc/dumpfs /etc/fsdump ; do
        if ls -dF $guess 2>&1 | grep '\* *$' > /dev/null ; then
                if test -n "`$guess W 2> /dev/null`" ; then
                        dump=$guess     # found it
.....
echo "DUMP_P=$dump      # BSD dump path: many variations" >> $tsaenv
```

Figure 4: Sample code from **saenv.setup** - BSD **dump**

```
:
# This is /etc/saenv which should be 444 root.  [ rest of header deleted ]
PATH=/usr/ucb:/usr/bin:/bin:/etc:/usr/etc       # exportable PATH
export PATH
.....
CRONCMD_B=false          # do we use the crontab command?
CRON_P=/etc/crontab      # path to real crontab file
CRON43_B=false           # is the crontab file in 4.3BSD format?
.....
DUMP_P=/bin/dump         # BSD dump path: many variations
.....
HOST=howdy               # the name of this host
CSH_P=/usr/bin/csh       # path to csh
ECHOPRE='-n'             # echo's no-newline prefix: '-n' (BSD), ''    (SysV)
ECHOSUF=''               # echo's no-newline suffix: ''   (BSD), '\c' (SysV)
PSALL_K=ax               # ps key to get all processes: 'ax' (BSD), '-e' (SysV)
```

Figure 5: Sample code from **saenv** (including code written by Figures 2, 3, and 4)

dump W to put something on standard out. It also expects System V dump to choke on the W flag and produce no standard output.

It turns out, however, that this isn't enough. Other code (not shown) has to deal with several situations. One vendor supplies a version of BSD **dump** which gives a "Segmentation fault" if **/etc/dumpdates** doesn't exist, so **saenv.setup** has to create one temporarily if necessary before running the above test. Another vendor supplies a version which requires a hyphen as the first character of the dump key, and even when fed a -W key only works if TERM is already set to a known terminal type. A third vendor supplies a special version of BSD **dump** which is meant to work with Exabyte drives, in addition to the normal BSD **dump**. Tracking down anomalies like these can make for frustrating coding.

```
:
# sahead - header for Data Center System Administration bourne-shell tools
# written by rca
#
# This file should 444 root, and should be sourced via ". /etc/dist/sahead".
# It is distributed automatically.
#
# The code includes lots of variable definitions for our
# System Administration scripts.

# untested PATH, /etc/saenv will fix later
PATH=/bin:/etc:/usr/bin:/usr/etc:/usr/ucb:/usr/amdahl/bin:/usr/lbin:/sbin
export PATH

# The variable names in all these scripts often use the following conventions:
#
# abbr  meaning                              example(s)
# XX_F  name of XX file                      CRON_F=root
# XX_D  directory where file XX lives        CRON_D=/usr/spool/cron/crontabs
# XX_P  full path to file XX                 CRON_P=/usr/spool/cron/crontabs/root
# XX_P  XX_P=$XX_D/$XX_F by definition       CRON_P=$CRON_D/$CRON_F
# XX_N  number                               ALIASRO_N=5 (rotations on "aliases")
# XX_K  key                                  PSALL_K=ax (BSD), PSALL_K=-e (SysV)
# XX_B  boolean                              CRONCMD_B=true if crontab command exists
# XX_C  command                              RSH_C=rsh (how do _you_ spell 'rsh'?)
#
# Remember - only the first 8 characters of a variable name are significant.
# When choosing variable names we have try to preserve the "_X" part.
#
# Many variables are set to null if they do not exist or not supported
# on the box.  For example, /etc/saenv would say "CSH_P=" on System V
# machines because they do not support csh.

## actual variable definitions
DTGT=/etc/dist          # distribution target, contains [links to] config files
DTGTBIN=$DTGT/bin       # distributed tools live here

.....
if there is a problem with /etc/saenv ; then    # pseudo code
        echo "some appropriate error message about /etc/saenv"
        exit 1
else
        . /etc/saenv  # we're fine so get local sysadmin environment
fi
```

Figure 6: Sample code from **sahead**.

Using The Research

For an example of how a tool might use the results of this research, see the **upd-dist** code in Figure 7. When a host runs **upd-dist**, it may get a copy of a new master **crontab** file. The master is in pre-4.3BSD format. By sourcing **sahead**, which in turn sources **saenv** (see Figure 6), **upd-dist** gets the information it needs to determine how to do the local installation. The `$CRONCMD_B` variable tells **upd-dist** whether the **crontab** command is required to install a new **crontab** file. `$CRON_P` is the full path to the **crontab** file. The `$CRON43_B` variable says whether to massage the new master into 4.3BSD format before installing it.

Common Tool Base

It would be impossible to create research code without a common tool set to write it in. Luckily, there seems to be one. Bugs notwithstanding, vendors seem to have maintained backwards compatibility with Version 7 usage and functionality of many of the common utilities in all the UNIX variants we have seen.

Shell scripts which run everywhere must be written with the greatest common denominator in mind. This means we can't take advantage of features available in many newer UNIX variants. We miss being able to use **sh** functions, **grep**'s "ignore-case" `-i` flag, **test**'s "executable" `-x` option, and the BSD **head** program, for example. But the Version 7 set of utilities is certainly rich enough to get the job done. An old manual is handy for Version 7 descriptions of commonly used tools such as **awk**, the **grep** family, **sed**, and **sh** [1].

Looking hazily into the future, the only potential addition to this toolset seems to be **perl** [2]. However, it would be a large amount of work to install and maintain **perl** on each of our UNIX variants. This alone probably means it won't be done. Furthermore, rewriting our current tools as **perl** scripts would also be real work.

```
:
# this is upd-dist - originally written by Diane Alter
. /etc/dist/sahead    # to get $DTGT and source /etc/saenv
.....
# if we got a new crontab file, install it
if test -n "`find $DTGT/crontab.dist -mtime -1 -print 2> /dev/null`" ; then
        current=/tmp/crontab.cur
        new=/tmp/crontab.new
        # get current version
        if test "$CRONCMD_B" = true ; then
                crontab -l > $current
        else
                cp $CRON_P $current
        fi
        # get new version, converting to 4.3BSD format if necessary
        if test "$CRON43_B" = "true" ; then
                $DTGTBIN/crontab.conv43 $DTGT/crontab.dist > $new
        else
                cp $DTGT/crontab.dist $new
        fi
        # merge new and current version, preserving local entries
        $DTGTBIN/cut-paste $new $current
        # install merged version
        if [ $CRONCMD_B = true ] ; then
                crontab -r
                crontab $current
        else
                cp $current $CRON_P
        fi
        # cleanup
        /bin/rm -f $current $new
fi
```

Figure 7: Sample code from **upd-dist**.

This seems to be a good place to say a few words about **perl**. *Programming perl* says that part of the process of installing **perl** is to "run a massive shell script called **Configure**, which tries to figure out everything about your system." ([2, p. 372]). This sounds like a very similar approach to the one discussed in this paper. Although **Configure** has been around a lot longer than **saenv.setup**, we were simply unaware of it when we began the development of this approach.

However, if installed everywhere, **perl** does provide the basics. It gives a common toolbase to write tools with, and also supplies the Configure script which could be used as a basis for writing a research tool such as **saenv.setup**. Even so, your tools must still research issues related to the organization of your site, and of course must still handle OS differences.

Developing Research Code

Figuring out how to code a test which will work everywhere can be easy. Sometimes it doesn't take much more than creating a list of possible answers, such as the code that builds the `PATH` environment variable.

Other times it is trickier. Finding the BSD **dump** program (as discussed above) is one example. Another is how to figure out what flag to give **ps** to get a listing of all the currently running processes. The code to determine this used to be easy.

```
if ps ax > /dev/null 2>&1 ; then
    PSALL_K=ax          # BSD key
else
    PSALL_K=-e          # SysV key
fi
```

This worked because versions of **ps** based on System V can be expected to give a failure exit status if handed the BSD-ish `ax` flag. But a new platform recently arrived with a System V-ish **ps** which accepted the `ax` flag too (without reporting all the processes, however). Now the code is more robust and looks like this.

```
npsax='ps ax 2>&1 | wc -l'
nps_e='ps -e 2>&1 | wc -l'
if test $npsax -gt $nps_e ; then ...
```

Before putting a chunk of code into **saenv.setup**, it needs to be tested widely. We maintain a list of roughly thirty hosts which represent more or less unique versions of UNIX. Small test scripts are created and tested on these representative hosts before the code is installed. One test run on these thirty hosts may take anywhere from ten to thirty minutes depending on the nature of the test and the problems encountered.

Although we have a fairly stable tool now, it is still occasionally necessary to add research topics to **saenv.setup**. If only one other tool might need additional research information, we make a judgement call about including it. If two or more tools need it, though, the decision to add is easy.

When a new type of UNIX system arrives, the easiest way to test **saenv.setup** on it is to simply run it and watch what happens. It is not uncommon to see unexpected error messages while it is running on a new system. The resulting **saenv** file needs to be checked for variables that do not have assigned values or other unexpected answers. Frequently, the master **saenv.setup** needs to be modified to handle the new situation.

Analysis of **saenv** files is made easier by having each host send a copy of **/etc/saenv** (along with other system configuration information) to a central location. Changes show up in a local newsgroup monitored by interested Data Center staff. Unexpected changes in a host's **saenv** file sometimes provides the first warning that a system is having problems. The entire set of **saenv** files can be processed by analysis tools to warn of problems or collect general information.

Writing and debugging research code is the most time consuming part of our approach. It is required whenever new research topics are added, it is common when a new UNIX variant arrives, and it crops up occasionally when bugs are discovered. Sometimes, the right solution to a new problem requires not only rewriting **saenv.setup** but also some of the tools which depend on it. Luckily, this work is a lot more fun and much less time consuming than our old haphazard methods. And the distribution of the new tools is automatic thanks to **upddist**.

Other Considerations

Where at all possible, values in **saenv** are unquoted.

```
PSALL_K=-e      # this is good
PSALL_K="-e"    # this is bad
PSALL_K='-e'    # this is bad too
```

We do this because some tools need to run remote **grep** commands to consult the **saenv** file on a remote machine. Interpretation of quotes can be a very tricky problem in shell coding, especially when using quotes in a remote shell command, and it is best to simply avoid the issue where possible. Unfortunately, if a string assigned to a variable contains a character which is special to the shell, like a backslash `\`, then some form of quoting is a must. This is the case with **echo**'s no-newline suffix for System V.

```
ECHOSUF='\c'
```

Tools which use **saenv** understand that if a variable is set to null, then it is not supported on the local host. For example, **saenv.setup** would put a line like this into **saenv** on a machine that did not have **csh**.

```
CSH_P=  # path to csh
```

Care must be taken when chosing names for variables set in **saenv** and **sahead**. Most of these variable names end in a `_X` suffix. The suffix describes the variable type, such as `_B` for boolean (See Figure 6 for more examples). This practice helps avoid conflict with names such as `HOME` or `SHELL` which are used by **sh** and other programs. Unfortunately, this also makes it harder to live within Bourne shell's limit of eight significant characters for variable names.

Some programs are especially likely to be renamed by vendors. Examples include the BSD **dump** backup program, and the remote shell program (frequently called **rsh** or **remsh**, though there are other variations). Some configuration files and programs are relatively frequent targets for being moved, such as the **aliases** file and **csh**.

Another Method Rejected

There is an alternative approach to making tools work everywhere, which we chose not to pursue. One could create a table containing everything you would need to know about all UNIX operating systems in your environment. That table could be distributed, along with a research tool to determine a) what OS is running on the machine in question, and b) truly local variations, such as the machine's hostname. Other tools could consult the research tool, and select the appropriate entry from the distributed table.

There are two disadvantages to this approach. One is that you would still have to do the research somehow to fill in the table. The other is that the table would be static and unexpected local variations might crop up. (This last might also be an advantage, in the sense that the table would enforce adherence to a standard.)

Advantages

In addition to the main goal of making our tools work as written everywhere, our approach has several other advantages.

- Improved User Support. It used to be hard for the Data Center to provide more than marginal support for weird workstations, which didn't really make anybody happy. This issue has largely disappeared since we can now offer more complete services on a much wider variety of platforms.
- Code maintenance. We only have to get **saenv.setup** working once. As we integrated **sahead** and **saenv** into our tools, the tools shrank because they no longer contained the research code.
- Performance. A tool can source **sahead** faster than it can do the research.
- Easily used by other interpreters. The Bourne shell code which sets variables in **sahead** and **saenv** files can be easily massaged into code suitable for sourcing by **perl** or **csh** scripts.
- Documentation. We document differences we encounter in the master version of **saenv.setup**, and to some extent in **saenv**. This gives us a centralized place to store information.

Drawbacks

The primary drawback to the approach we've taken is a typical centralization problem. Each host has many of its system administration eggs in the **saenv.setup/sahead/saenv** basket. When something goes wrong here, many tools can break in interesting ways. That's why it is so important to do the code testing on the representative hosts.

Also, we are keeping our fingers crossed that no vendor will use the pathnames **/etc/dist** or **/etc/saenv** in their UNIX variant. If it happens, though, we'll simply rename our files and go on our merry way.

Future Plans

By the time this paper reaches publication, we expect to have created an **salib** ("ess-ay-lib") shell script to be added to the list of distributed files. Like **sahead** and **saenv**, it will be sourced by other tools. It will contain a library of useful code chunks and macros. For example, it will contain a `NEED` macro, so a tool could do something like

```
NEEDLIST="CRON_P CRON43_B CRONCMD_B"
eval $NEED
```

Evaluating this macro would cause the calling script to quit with an appropriate error message if any of these variables had unreasonable values. Boolean variables would be expected to be true or false, path variables would be expected to contain slash characters, and so on.

Conclusions

We should have done this a long time ago!

Acknowledgements

The basic idea for this approach came out of a conversation I had with my fellow system administrator Sid Shapiro. Thanks are also due to other coworkers who have put up with problems caused by the "implementation details". The **upd-dist** script was originally written by Diane Alter.

Author Information

Bob Arnold is a Systems Administrator for the Data Center group at the Ingres Products Division of Ask, Incorporated. He has six years of experience in the field. Bob plays in the "Insensitives", who's motto is *Rock & Roll Without Shame!!!* He can be contacted via U.S. Mail at ASK/Ingres Products Division, 1080 Marina Village Parkway, Alameda, CA, 94501. Send electronic mail to rca@Ingres.com .

References

[1] Bell Telephone Laboratories, Inc., *UNIX Programmer's Manual*, Seventh Edition, Vols. 1& 2, New York, NY: Holt, Rinehart and Winston, 1983.

[2] Larry Wall and Randal L. Schwartz, *Programming perl*, Sebastapol, CA: O'Reilly & Associates, Inc., 1991.

Software Maintenance in a Campus Environment: The Xhier Approach

John Sellens - Math Faculty Computing Facility, University of Waterloo

ABSTRACT

Xhier is a system for software maintenance and distribution, currently in use at the University of Waterloo. It allows easy, automatic software installation on a variety of UNIX systems. This paper describes some of the design goals of **xhier**, its structure and operation, as well as some of the problems we have encountered, and some future goals.

The Motivation for Xhier

The Math Faculty Computing Facility (MFCF) is a major UNIX support organization at the University of Waterloo, and, in addition to other responsibilities specific to the Math Faculty, provides software support for UNIX machines all over campus using the **xhier**[1] software maintenance and distribution system.

MFCF started doing UNIX support before networks and workstations were so widespread. It was not too many years ago when the MFCF UNIX environment was a single DEC VAX 11/780 running 4.2BSD UNIX. In that environment, adding and updating software was relatively easy, since operating system releases were infrequent and the amount of third party software was small. It was easy to just add some source to `/usr/src` and install some commands in `/usr/bin`. If there were bugs in the OS that needed fixing, or enhancements that were desired, it was a (relatively) simple matter to modify the BSD source, and install the change. A few more BSD VAXes were added without too much pain, and careful use of tools like *rdist*(1) and *rcp*(1) made it possible to keep things up to date.

Once UNIX machines started being more common on campus, and different models and brands of machines started appearing, it quickly become obvious that a different approach was required. With operating systems being updated as often as every few months, different operating systems using different file system organizations, and with useful third party software proliferating, a more structured and automated approach was needed.

Development of the **xhier** system was started in early 1989 and has been ongoing ever since, with many enhancements and fixes still planned. MFCF currently provides software support for 11 different machine "architectures", using the automated tools provided by **xhier**. **Xhier** is currently used by over 250 machines, with over 1200 commands in more than 200 software "packages" available, with over 14,000 package installations on the various machines.

The Problem to be Solved

MFCF was faced with the task of providing software support for multiple architectures, serving multiple machine administrations, and different user groups and purposes. A research machine in the Computer Science department might have different needs than a student workstation cluster in Electrical Engineering, though both would want some form of software support, and some selection of locally installed commands. Additionally, most users want a consistent environment, regardless of the architecture and operating system of the machine they currently happen to be using.

Accordingly, it was necessary to find some way to distribute software to, and make it available on, large numbers of different machines. This had to be done with a large amount of configuration flexibility (so it could be tailored to the needs of different machines and administrations), and it had to be as automatic as possible, in order to make the most efficient use of the software support staff.

Faced with this environment, MFCF's work on **xhier** has proceeded with these primary goals:

- to simplify the distribution of updates,
- to simplify the installation (on multiple architectures),
- to facilitate sharing software via remote file systems, and
- to minimize changes and additions to the file structure distributed with operating systems.

An important underlying goal is that of automation of as many aspects of the software maintenance and distribution process as possible.

Xhier takes a different and more automated approach than systems like the **depot** [1]. Appendix B discusses some of the differences between **xhier** and the **depot**.

[1] The name **xhier** is derived from the BSD *hier*(7) "file system hierarchy" man page, where the "x" prefix indicates eXperimental, though eXtended, eXtravagant, and eXhausting have all been offered as alternatives.

The Overall Structure

In order to minimize the changes to stock file organizations, it was decided to locate **xhier**'d software in a separate (hopefully) unique location - under `/software`[2].

Under `/software` are the package directories. In **xhier**, software is organized into "packages" of related software. For example, there is a **tex** package, containing TeX related software, and a **gnu** package containing GNU software. Separating software into separate packages has two main advantages: it allows a system administrator to pick and choose what software to make available on a machine (which may depend on licensing or size considerations), and it allows the grouping of related software for maintenance and similar purposes. It also allows multiple versions of the same software to be installed and available at the same time.

The other goals are approached using the details of the organization of software packages, and by the use of support programs.

Xhier recoginizes that there are three main steps in making software available to users:

- The files must be placed in an appropriate location, and be compiled if necessary.
- The software must be configured and initialized appropriately.
- The software must be made available to the users (typically through search rules).

The structure and tools of **xhier** are intended to make these steps as easy as possible.

```
% lc /software/x11
Directories:
.admin  bin   data   doc   export
include lib   logs   man
```

Figure 1: A Typical Package Directory

Software Package Organization

Each software package contains all the files associated with that software, along with other files required for the distribution and installation of the package. A package typically consists of a number of sub-directories, with the common ones being:

- `bin`, `maintenance` and `servers` contain commands for users, system and software maintainers, and other programs to use,
- `lib` for libraries,
- `man` for man pages, and
- `data` for data and configuration files.

The structure and contents of a particular package are determined by the needs of the package, but directories like `bin` and `man` are used by other programs to make package components available to users (more on this below). Figure 1 shows a typical package directory.

Two sub-directories are used by **xhier** for distribution and installation.

```
% lc /software/x11/export
Files:
boottime   crontab    inetd.conf
services
```

Figure 2: A Typical Package `export` Directory

The optional `export` directory (Figure 2) contains information to be used outside the package. It contains such things as `services` and `inetd.conf` file entries required by the package (e.g., if the package contains a network daemon), commands to be run by *cron*(8), commands to be run on system startup, as well as information on userids and groups required by the package, and patches to standard system files like `/etc/rc`. These file fragments and patches are applied automatically to the system configuration files when a package is installed.

```
% lc /software/x11/.admin
Files:
Dependencies Install      Maintainer
Options      Targets      file-types
```

Figure 3: A Typical Package `.admin` Directory

Each package is required to have a `.admin` directory (Figure 3), which contains information used in the installation and distribution of the package. It typically contains 4 files:

- `Maintainer`, which gives the mail address of the person responsible for installing and maintaining the software,
- `Dependencies`, which lists other packages required for proper operation of the package (e.g., a package might make use of commands provided by other packages),
- `Options`, which sets various options that determine how the package is distributed and installed, and
- `Install`, a program (typically a shell script) that performs any package specific initialization to prepare the package for use on a particular machine (e.g., initialize configuration files, check for errors, replace stock files if necessary, etc.).

A couple of other files (`Targets` and `file-types`) are sometimes used to influence the distribution of a package[3].

[2] `/software` is typically a symbolic link to some appropriate location, but references to **xhier**'d software use the standardized name `/software`, rather than the actual location of the directory on a particular machine.

[3] These files are somewhat of a kludge and may disappear in the future.

The package structure makes each package a separate entity, containing all the knowledge required to install and use the software it contains. It also provides a consistent structure so that packages tend to look very similar, which makes maintenance easier.

See appendix A for an example of how packages are created and installed.

Complicating the Structure

Now for the complication. Recent versions of UNIX have reorganized parts of the file system to make file sharing easier. For example, SunOS has `/usr/share`, which contains those files that can be shared among all machines, regardless of CPU type. **Xhier** takes this idea a step (or several steps) further.

Separating files based on their type has two main advantages, both of which help to satisfy the goals of **xhier**. It makes it easier to share software via remote file systems (e.g., NFS), because you can share as much of a package as possible, and minimize duplication across machines of the same or different hardware architectures. And it makes software distribution easier because it makes it easier to use a tool like *rdist*(1) to distribute the appropriate file types to the appropriate machines e.g., shared files like man pages can be distributed everywhere, while executable binaries can be distributed to just those machines with the same architecture.

Xhier currently recognizes six file types:

- **share**: files like man pages, shell scripts, and (typically) the `.admin` and `export` directories
- **arch**: files that are different on different hardware/operating system combinations, most commonly executable binaries
- **spool**: files of a transient nature, like print requests
- **local**: files specific to a particular machine, such as local configuration or log files
- **regional**: files shared between clients and their password and home directory server, such as nntp server names
- **admin**: files shared among all machines in a particular administration, such as printer permission files

The **regional** and **admin** types are variations on the **local** type that seem to be useful. In practice, it seems that there are actually more file types (e.g., **man** might be used to describe man pages and documentation, because you might not want those locally on *every* machine, in order to save space), but these six types are recognized explicitly in the **xhier** file structure.

This is accomplished through the use of the `/.software` directory. `/.software` contains one sub-directory (or, more typically, a symbolic link to, or an NFS mount of, an appropriately located directory) for each of the six basic file types, with each sub-directory containing package directories. For example, the man pages for the **tex** package are in `share/tex/man` in the `/.software` directory, while the command binaries are in `/.software/arch/tex/bin`. Note that the two directories `/software` and `/.software/share` are the same place[4].

Every package has a shared component; in particular, the `.admin` directory must exist, and is always a shared directory. And since `/software` and `/.software/share` are the same place, this ensures that, for a package called **pkg**, `/software/pkg` will always exist.

It is desirable to hide the file type structure of a package from the outside world, so that a user or another program does not need to be concerned with the internal structure of a package. So, even though every file in a package can be referenced through `/.software` type directories, references from outside the package itself are always made through `/software/pkg`. (A package is allowed to know its own structure, and refer through `/.software` as appropriate.) This is accomplished by the (liberal) use of symbolic links in the `/software/package` directory. For example, `/software/tex/bin` is actually a symbolic link to `arch/tex/bin` in the `/.software` directory. It is also possible to have a shared directory contain real files along with symbolic links to files in directories of other types. For example, a package might have commands that are primarily shareable shell scripts, with the few compiled commands being referenced through symbolic links in the shared `bin` directory[5]. Note that these links, while pointing to different absolute locations on different machines, are shareable, since they link through `/.software`.

The result is that the file type directories under `/.software` provide separate hierarchies for different types, which makes file sharing and distribution much easier. For example, diskless workstations will usually share and mount all file types from their server, with the exception of the `local` type, which each client will have its own

[4]This means that one of `/software` and `/.software/share` is redundant, but `/software` was the original name, and provides a nice name for people to use, while `/.software/share` was included primarily for completeness.

[5]This latter structure, while within the rules, can sometimes be confusing, so it is usually better to make the directory the symbolic link, and its contents be real files, rather than a mixture of the two.

version of. The `/software` link provides a naming convention for access to the components of a package. Note that a given machine will have the correct type hierarchies for itself only, e.g., `/.software/arch` on a VAX contains only VAX binaries, and on a Sun will contain only Sun binaries.

Automating Everything

Xhier is built on the idea that automation is good. Accordingly, there is an **xhier** program to do just about anything to do with software installation and distribution.

There are currently about 80 commands in the **xhier** package, most of them shell scripts. Commands are used for such things as package creation, maintenance and installation, and distribution, along with some utilities that are useful for day-to-day operation (log file rollers, etc.). The following is a summary of some of the more prominent commands.

The *xh-mkpkg* command creates a package skeleton in the `/software` directory, with command and support directories, and some template files, that can be filled in by the package creator. This makes the initial creation of a package simpler, because a package creator doesn't have to remember all the details. *xh-make* is used as a cover for *make*(1) and *imake*(1) and uses a standard *imake* template to make it easy to compile and install a program (see Figure 4). *xh-make*, when used with an `xh-imakefile`, ensures that any architecture-specific libraries and commands are used.

```
#include "../PackageName"
Program(callpat,c,user)
```

Figure 4: Typical `xh-imakefile` file for use with *xh-make*

Once a package has been compiled and installed into the package directory, the *xh-install* command makes it available for use. It does this by running the package's `Install` script, to do any package-specific tasks (such as package configuration, or replacement of any stock files), and then, if that is successful, invoking other programs to patch system files (like `/etc/services`) from the fragments in the package's `export` directory.

Once a package has been installed, and is working on one machine[6], it can be distributed to, and built on other machines. This is a two-step process. The *xh-distribute* command is used to distribute the package contents and structure to other machines. It creates a `Distfile` for use with **rdist** that sends the appropriate files to the appropriate machines lower on the tree. For example, it will send the shared files everywhere, but architecture specific files will only be sent to machines of the same architecture (though it will complain if files on a machine of a different architecture are out of date, or missing). *xh-distribute* then invokes **rdist** and summarizes the output for easier human consumption.

The second step is *xh-sdist*, which distributes the source for the package to machines of other hardware/software architectures (the "architecture master" machines)[7], and runs *xh-make* on those machines in the background, collecting the output and mailing it to the invoker. *xh-sdist* has a number of options and configuration files to control exactly what gets distributed to the remote machines (e.g., `.o` files are not distributed). *xh-sdist* has turned out to be an incredibly useful tool - it makes updating software and installing fixes almost trivial[8].

To help ensure that software is kept up to date, working, and consistent on all machines, *xh-maintenance* is run weekly from the *cron* (on every **xhier**'d machine). *xh-maintenance* does some housekeeping and consistency inspection, and runs *xh-install* and *xh-distribute* on all packages, arranging to send the appropriate output to the appropriate package and system maintainers.

Since *xh-maintenance* is run weekly, starting at the same time on every machine, it can take several weeks for package updates to reach the bottom of the distribution tree[9]. And so the other important distribution-related tool is *xh-dist-recurse*, which attempts to distribute a package recursively down the distribution tree. This can be very useful in distributing fixes quickly, in case you've inadvertently broken something and don't want to wait for the usual propagation delays to distribute the fix.

[6]Currently, this must be a machine higher up the tree, usually the one central machine, but this will change in the future, so that any machine can be the initial "maintenance" machine for a package.

[7]*xh-sdist* sends the source to a temporary directory on the remote machines, and it is automatically removed after a few days.

[8]It should be pointed out that *xh-sdist* is somewhat of a kludge, because it introduces an arbitrary distinction to some machines, that of being an architecture master, when it should be possible to distribute source to and build a package on just about any machine.

[9]This propagation delay can be a blessing, because it can keep broken packages from infecting the entire distribution tree.

Making it all Available to the Users

In a conventional software installation, software is installed in one location only (commonly `/usr/local/bin`), or in other OS-specific directories (such as Sun's `/usr/openwin/bin`), which the user is expected to add to `PATH`. Expecting a user to know where software is installed is not always reasonable. For many years, MFCF has had a command, **showpath**, to help users set their `PATH` without having to know the specifics of a particular machine.

With **xhier**, once the packages are installed, there are many different package directories, each containing command directories (e.g., `bin`), `man` directories, `include` directories, and/or `lib` directories. The **showpath** command could simply produce a list of all the `bin` directories in all the installed packages. This, however, quickly becomes too cumbersome to handle - a 3,000 character `PATH` containing 200 directories is a little large. It also becomes inefficient since some shells hash the `PATH` contents, but often only the first few elements (csh apparently hashes the first 8 elements of `PATH`).

Xhier approaches this problem by creating a directory that contains a link to each of the installed, packaged commands, using the *xh-make-links* command. These links have to be symbolic links, since the commands are not always going to be on the same filesystem (or even the same machine). There are actually three of these directories, one for user commands, one for maintenance commands, and one for commands used by other programs (servers). These directories are system-wide and system-specific in that they contain links to the commands for only those packages that the system administrator wishes to be "default" packages[10]. A user's `PATH` consists of the directory of links, the appropriate stock directories (for that architecture), and any other directories the user wishes to include. These directories of symbolic links might be expected to cause a performance penalty on command invocation, but, in practice, it seems not to be a problem.

This "searchrules" approach to commands works well, but it is harder to apply it to the other parts of a package. Many *man*(1) commands understand a `MANPATH` variable, but most compilers and loaders don't have an easy way (i.e., an environment variable) to extend the searchrules for include files and libraries. Include files and libraries are linked (again using *xh-make-links*) into the standard locations honoured by the OS, usually `/usr/include` and `/usr/local/lib`.

One more minor complication. MFCF had been using the *rman* remote man command[11] to provide man pages to smaller machines, where there is often not enough disk space to keep man pages locally. Local modifications to *rman* allow the remote man page server to deal with packaged man pages by special interpretation of the `MANPATH` variable. Figure 5 shows typical commands that would be included in a `.cshrc` file, and common values of key environment variables.

Some Lessons Learned from Xhier

Xhier tends to point out some of the more obscure points of software installation. One prime lesson is that doing software distribution can turn out to be a complicated process, with a certain amount of overhead.

Automation is a necessity, once more than a few machines are involved, but it turns out that some things can be hard to automate. For example, it's not too hard to add an entry to the `/etc/services` file, but it's a little harder to notice conflicts or replacements, and some packages might want a particular userid to exist (e.g., "games"), but it's hard to add an appropriate account to a `passwd` file automatically. The current **xhier** utilities handle *inetd*, *cron*, and `/etc/rc` entries fairly well (by automatically editing the appropriate system files, and disabling

[10] In practice, the set of "default" packages is virtually the same everywhere.

[11] By Jonathan C. Broome, posted to `net.sources` in 1985.

```
setenv PATH     'showpath usedby=user $HOME/bin standard'
setenv MANPATH  'showpath class=man standard'
setenv EDITOR   'showpath findfirst=vi'
% echo $PATH
/u/jmsellens/bin:/.software/local/.admin/bins/bin:/usr/ucb:/bin:/usr/bin
% echo $MANPATH
/software/.admin/man/:/usr/man/
% echo $EDITOR
/.software/local/.admin/bins/bin/vi
```

Figure 5: Typical extract from a user's `.cshrc` file, and resultant variables

conflicting stock entries), but `passwd` and `group` file entries still have to be applied by hand (local administrators are prompted to do it when a package is installed). And while automation allows easy installation and distribution, it also makes it easy to distribute broken software and automatically destroy dozens of machines.

Some software comes with particular pathnames built-in, often hard-wired pathnames are sprinkled throughout the code (especially files destined for `/etc`), and it's often hard to change these pathnames to refer to locations under `/software` instead. In other cases, they are easy to change, but the same symbolic name is used for different kinds of files. For example, the *make* variable "USRLIB" might be set to `/usr/lib`, and be the location for both object libraries and data files, or "ETCDIR" might be used to contain local configuration files and system maintenance commands. Files are often named for their location, rather than for their function. **Xhier**-ing a package forces an installer to interpret the given pathnames, and choose appropriate **xhier** pathnames instead.

In addition, certain pathnames, like `/usr/lib/sendmail` and `/usr/ucb/lpr`, are known to many programs, and so any replacement versions of these programs must also replace the well-known stock pathnames (with links to **xhier** file locations).

It's hard to organize NFS (or, presumably, other remote file system) mounts to make software sharing foolproof. This problem is especially evident with symbolic links. A symlink to an absolute pathname will start looking for the file at `/` on the local machine, even though the link is on the remote machine. And conversely, a relative symlink will tend to stay on the remote machine, unless it passes up through a mount point. Some uses seem to call for absolute symlinks, and some seem to call for relative symlinks, but so far we have been unable to come up with a rule (or even a guideline) that works in all cases.

Choosing what goes in what package is hard. It's not always easy to group software into appropriate packages, and different administrators might want different parts of a package, instead of the whole thing. Large packages are easier to deal with, but small packages give greater flexibility.

It's easy to generate lots of junk mail, so efficient ways of summarizing and distributing it are required.

Future Plans and Dreams

The current distribution mechanism uses a tree structure, with each member in the tree "pushing" packages to lower nodes in the tree. This is less than optimal, but was necessary since the only distribution tool available at the time of implementation was *rdist*. Some of the problems with this approach are

- Distribution is scheduled by the upstream host(s). A subsidiary node has (essentially) no control over when its software gets updated. This is annoying when a guaranteed stable environment is desired, such as when one is trying to finish a thesis, or a machine is used for student classwork. In practice, this hasn't turned out to be a problem, since a request to an upstream administrator is usually enough to modify the distribution list.
- It is inconvenient to have two (or more) distinct machines be the "masters" of two different packages and exchange packages with each other, because the use of *rdist* forces essentially "wide-open" access for the upstream machine on the downstream machine (via `/.rhosts`).
- This style of distribution seems to require wide open access to a machine by its distribution machine.
- Its hard for a given machine to dictate exactly what software it gets, because that is decided by the upstream host(s).

For these reasons, distribution control will pass to the receiving machines, and the tree structure will be replaced with a more flexible one, where any two cooperating machines can exchange whatever software they wish to.

This, however, requires a program like *track*(1) that can do most of the same things that (the locally enhanced) *rdist*(1) can do. This work is currently underway. It also requires fairly major changes to the mechanisms that determine which parts of which packages get sent to which machines. These changes are almost in place, to be used with the current distribution mechanism until something "track-like" is ready.

Most machines don't have enough local disk space to contain all packages, or even just the ones they want to use. Current space usage on the primary maintenance machine[12] is

```
% du -s /.software/*
105     /.software/admin
272164  /.software/arch
11823   /.software/local
727     /.software/regional
231340  /.software/share
```

or a total of over 500 megabytes, in addition to the space required by the OS[13]. As a result, remote file system mounts (currently only via NFS) are

[12] "watmath", a MIPS RC6280.

[13] This excludes the spool files (especially news articles) under `/.software/spool`.

required to make the necessary software available on many machines. This is usually done in bulk, where a machine might mount all of `/.software/share` and `/.software/arch` from a willing server. Things get complicated when you wish to have some packages locally and some remote mounted, since you either have to mount each package individually, or arrange that `/.software/*/package` are either real directories or symbolic links through the appropriate mount points.

At present, *all* remote mounting is done manually, which is getting less and less practical. Some method needs to be devised to automatically make the appropriate mounts and links, which will allow a local administrator to easily add a software package to a machine. It's not at all clear what the best approach to this problem is.

Xhier-ing a software package forces a software maintainer to look beyond the local machine, and consider how a package would be installed and used on different machines, with different administrators, perhaps wishing different defaults or environments. However, the current outlook tends to stop at the edges of the campus. A ''shared'' file tends to mean one that applies to the campus as a whole, rather than to the ''world''. In fact, there is currently no way in **xhier** to indicate a file with type ''campus'', i.e., there is no `/.software/campus`. It would be nice if it was possible to distribute **xhier** and **xhier**'d packages to other organizations, but that brings other problems to light, such as the necessity to trust a remote software provider.

Conclusions and Observations

With **xhier**, it is now almost trivial to make a new software package available to hundreds of machines. It's easy to make and distribute fixes or enhancements.

Xhier allows MFCF to provide software support, and a consistent user environment, on many more machines than would be otherwise possible.

Xhier does have a number of faults

- It is large, and still changing, and it is not always easy to understand how all the parts fit together. However, most software maintainers do not need to know how it works, just how to create a package and make it available.
- It tends to create a maze of symbolic links.
- Requires a fair amount of machine overhead to keep it shuffling software around.
- It relies on having a fairly BSD-ish set of system services.

but overall it tends to meet the needs. For MFCF and the University of Waterloo at least, **xhier** is a success.

Appendix A: Creating and Installing a Package

Most software packages are very easy to create, though large pieces of software, like *gcc* or the X11 software, are more complicated. Typical package creation and installation goes something like the following example installation of package **foo**.

First, the source is unpacked into `/usr/source/foo`, and the `Makefiles` are modified as appropriate, to set the right options, and installation locations. Many times this is done using an `xh-imakefile` (see Figure 4 for an example) that is processed with *xh-imake* to allow easier per-architecture customizing.

The command

```
% xh-mkpkg foo
```

is used to create a skeleton structure in `/software/foo`. This structure is then adjusted by hand as required. Usually this just means removing the parts of the skeleton not required by this package.

Then the files in the `.admin` directory are edited as appropriate. The appropriate options are set from the defaults in the `Options` file, and the `Install` script is modified to do the appropriate things on installation and uninstallation. Often this means just ending up with the script

```
#!/bin/csh -f
exit 0
```

The **foo** package name is then ''registered'' by adding it to the *software* (i) man page[14].

The software is installed via

```
% xh-make install
```

in the source directory. Once that completes, the commands

```
% xh-install foo
% xh-distribute foo
% xh-sdist -m install
```

install the package locally, and then distribute the package structure and source to the other architecture master machines, and compile and install the package there.

Once this has been done, the **foo** package is available to any **xhier**'d machine on campus, with very little effort.

[14] This step needs a little refinement.

Appendix B. Comparison to the Depot

The **depot**, as described in [1], has some similarities to **xhier**, but is less automated, and is based on a client-server distributed file system environment, rather than **xhier**'s greater emphasis on the use of local file space to hold packages.

Both systems separate software into packages, and define a structure for the individual packages. The **depot** requires human intervention for each installation, in order to execute an optional installation script, and to modify system files (such as `inetd.conf`) if necessary; **xhier** uses the *xh-install* program and the package's `Install` script. The **depot** uses redundant servers for the packages, but distributing a package to a redundant server is done manually; **xhier** has automatic distribution of the packages. The **depot** is geared primarily to the use of server machines that contain all the files for all architectures. This makes it less useful for standalone machines (such as a home workstation), since such a machine would have to have the binaries for every architecture, requiring more disk space (unless manual pruning was used).

Maintenance activities are less automated in the **depot** as well. In order to recompile or update a package, a maintainer has to go to the **depot** administration machine of the appropriate architecture, and compile and install the package there. **Xhier**, on the other hand, uses the *xh-sdist* command to rebuild and install a package on all subsidiary architectures, which helps ensure that packages are kept up to date on all architectures.

Since the **depot** uses *automount* mount points, as listed in an NIS database, to make packages available to client machines, it is harder for a system administrator to tailor the set of packages available on a particular machine or set of machines. **Xhier** makes it easy for an administrator to pick and choose what packages are available on his or her machine(s). Admittedly, in a primarily client-server, workstation environment, this is less of a problem.

With **xhier**, a system administrator can control when packages get updated on his or her machine, which is useful in a student lab environment, or on a personal workstation when trying to complete a project. Since the **depot** is based on the client-server model, it is much more difficult for a client to "refuse" an update.

Xhier provides a set of tools for modifying, or replacing vendor provided files, which makes it possible to correct flaws in stock systems, or customize them to local requirements. **Xhier** provides simple interfaces to system services such as `/etc/rc` and *cron*. This means that a package maintainer does not usually have to worry about system-specific details.

Xhier makes extensive use of search paths in order to hide the details of an implementation from the users. With the **depot**, a user must know where programs and man pages are located in the hierarchy in order to make use of them.

Xhier Availability

Xhier is still under development, and still contains many transitional provisions to correct past mistakes, and it is currently difficult to bootstrap **xhier** into a new environment (such as another campus). It is also a large collection of source, which relies in part on modifications to licensed software.

As such, **xhier** is not currently available for distribution, though we hope to be able to make all or part of it and its tools available at some point in the future.

Author Information

The work described in this paper is due to the efforts of all members of the Math Faculty Computing Facility software group, although the paper itself was compiled by John Sellens, currently with the Department of Computing Services at the University of Waterloo, in the UNIX Support group. Any and all errors, confusion and inaccuracies should be attributed to him. Reach him via physical mail at University of Waterloo; Waterloo, Ontario; N2L 3G1 CANADA. Reach him via networked electronic mail at jmsellens@watserv1.uwaterloo.ca.

References

1. Manheimer, K., B. A. Warsaw, S. N. Clark, and W. Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries", *LISA IV Proceedings*, October 1990, pp. 37-46.

Integrating UNIX Within a Microcomputer-Oriented Development Environment

Peter Bumbulis, Donald Cowan, Eric Giguère, Terry Stepien – University of Waterloo

ABSTRACT

The Computer Systems Group (CSG), a software research group at the University of Waterloo, maintains a number of different computer systems within the University's internetwork. These computing facilities include four UNIX-based systems, an IBM 4341 mainframe and several dozen microcomputers of various types. Four different physical networks connect the systems to each other and to the campus internet.

Of the many projects currently underway at CSG, most involve the use of microcomputers. Apart from the projects that directly involve UNIX or the X Window System, UNIX is used mostly for support services (such as mail, news, printing, backups and name service) by both technical and administrative staff.

This paper discusses our experiences in integrating UNIX within the CSG environment to provide those services and to make them accessible from any microcomputer. Special attention is given to the networking of these diverse systems.

1. Introduction

UNIX-based systems have been gaining in popularity but are still overshadowed by the millions of microcomputers in daily use. As a consequence, many organizations find themselves supporting at least two distinctly different computing platforms on separate networks, each with their own strengths. It would be useful, then, to find ways to integrate the systems and share their mutual benefits.

The Computer Systems Group (CSG) at the University of Waterloo is a research group working with a variety of software and hardware platforms. Members of CSG, including support and administrative staff, have exclusive access to at least one microcomputer or workstation. Because of this variety, the CSG's integration efforts have three main objectives:

- *Complete connectivity*. Each microcomputer/workstation is able to access any computer system connected to the CSG internal network or the University campus network.
- *Availability of services*. Useful services (mail, printing, file access, backups) are available from all microcomputers/workstations.
- *Preserving a familiar environment*. Both technical and non-technical users can access services using simple procedures and familiar interfaces.

UNIX is a significant component in supporting these objectives.

2. The Computing Environment

Current hardware at CSG includes three Sun workstations, an AIX-based system, an IBM 4341 mainframe, and a mixture of over three dozen Macintoshes and PCs including IBM PCs and PS/2s, and compatibles. Projects underway involve VM/CMS, UNIX, the X Window System (both Motif and Open Look), Microsoft Windows, and the Macintosh.

The University's computing environment includes a cluster of IBM 4300-series mainframes, over 200 UNIX systems, and numerous local area networks (LANs) connecting PC and Macintosh microcomputers. A campus-wide TCP/IP-based Ethernet (with a gateway to the national TCP/IP network) links most machines. An early version of the University network is illustrated in [COWAN88b].

3. Making the Connection

Connectivity has been an important goal for CSG since the early 1980s, and the physical network has evolved with the hardware technology. As part of this goal CSG has developed a local area network (LAN) for IBM PCs, PS/2s and compatibles, and one for Macintoshes. These two LANs are called JANET and MacJANET [COWAN88a] respectively; PCs on JANET communicate with our own JANET protocols while MacJANET uses AppleTalk. JANET and MacJANET support print-servers and disk- or file-servers, and currently operate on a number of different physical network layers including Ethernet, LocalTalk, Token-Ring, and PC Network Baseband.

The existence of the JANET and MacJANET LANs simplified the complete connectivity objective, since each microcomputer already had a network connection (including necessary hardware). Coexistence of these two LANs and integration with other LAN technologies were significant components of this goal.

3.1 The Physical Network Environment

Currently CSG maintains a set of four physical interconnected networks: Ethernet, LocalTalk, Token-Ring, and PC Network Baseband. A diagram illustrating the interconnection of the four physical networks and external networks is shown in Figure 1 below.

The Sun workstations are connected directly to the Ethernet, while the Macintoshes are connected to the Ethernet or LocalTalk network. Four PostScript laser printers (one supports colour) are also connected to the LocalTalk network. The PCs are attached to either Ethernet, Token-Ring, or PC Network Baseband and the AIX machine and the IBM 4341 are on the Token-Ring network. The three JANET servers shown in Figure 1 in reality represent one server which is connected to all three networks. We have also experimented successfully with connections to remote LANs using serial lines and these are shown at the top of Figure 1.

Why does CSG have four physical networks? Ethernet is a significant component of the University network and is the intended direction for new or replacement microcomputers/workstations, while LocalTalk is the default network interface for many models of the Macintosh microcomputer. Token-Ring provides a convenient method for connecting microcomputers/workstations to IBM mainframe systems. PC Network Baseband was one of the earlier microcomputer network interfaces and is currently used on many of our existing machines.

3.2 The Logical Network Environment

Three different logical networks use the four physical networks. These networks, called the T-Network, A-Network and J-Network, are shown in Figures 2, 3 and 4. The remainder of this section is devoted to describing our approach to constructing these three logical networks.

3.2.1 The T-Network

The T-Network shown in Figure 2 carries IP datagrams and allows communication among machines running TCP/IP. Hosts on the T-Network include PCs running DOS, a PC running AIX,

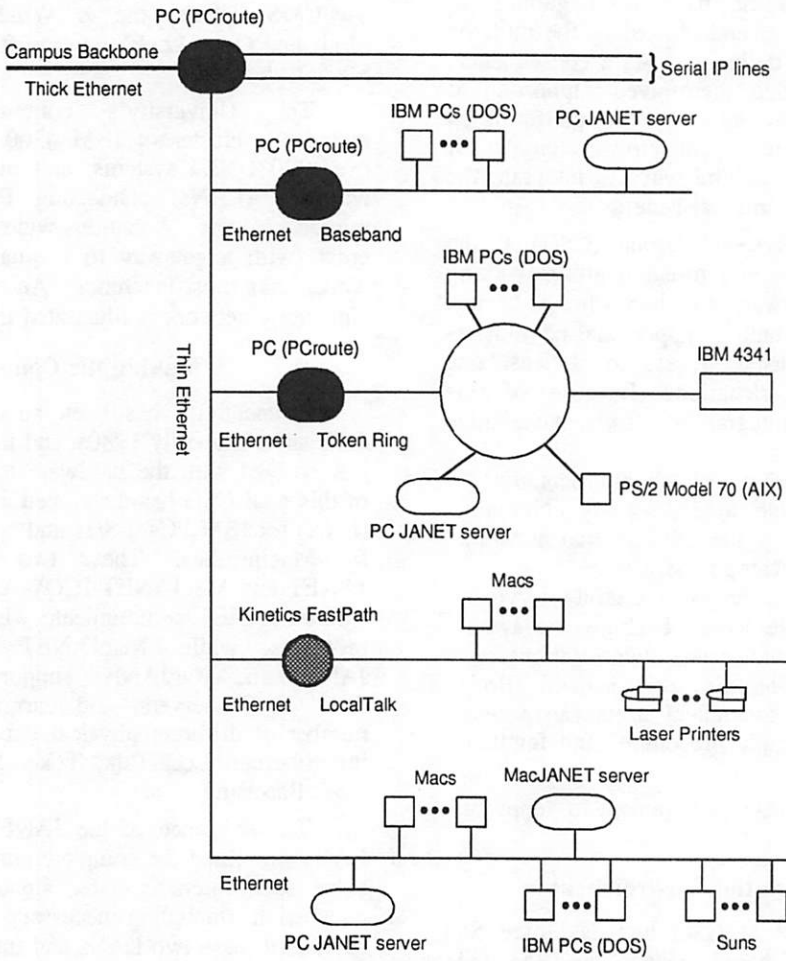


Figure 1: The CSG Physical Network

Macintoshes, Sun workstations, and an IBM 4341 running VM/CMS.

PCs configured as routers and running PCroute transfer IP datagrams between the Token-Ring, PC Network Baseband, Ethernet, and external networks. A Kinetics Fastpath Ethernet bridge/router transfers IP packets between the Ethernet and LocalTalk networks. Note that the IP packets on the LocalTalk network are actually encapsulated in LocalTalk packets. Services available on the T-Network include mail, printing, file sharing and backups.

3.2.2 The A-Network

The A-Network connects machines communicating with the AppleTalk protocol. Hosts on the A-Network include Macintoshes, LaserWriters, and a Sun workstation. The Kinetics Fastpath Ethernet bridge/router transfers AppleTalk packets between the Ethernet and LocalTalk networks. On the Ethernet, the Kinetics Fastpath can handle both "raw" AppleTalk packets (EtherTalk phase I and phase II) and AppleTalk packets encapsulated in IP datagrams.

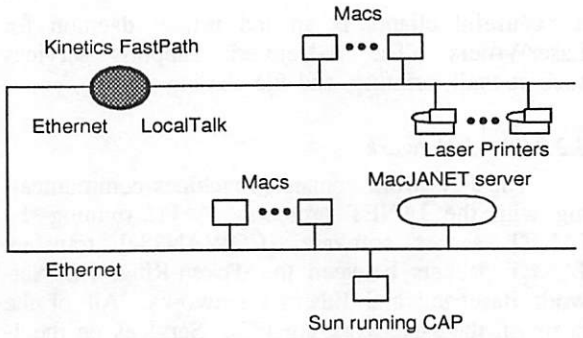


Figure 3: The A-Network

IP encapsulated AppleTalk packets are primarily used by Unix systems communicating with TCP/IP. Encapsulating AppleTalk packets in IP allows these systems to connect to the A-Network without kernel modifications, or the need for a low level network device interface. The Columbia AppleTalk Package (CAP) provides the AppleTalk support for the Sun workstation. As well as providing low level protocol support, it implements a variety of AppleTalk clients and servers. One of the

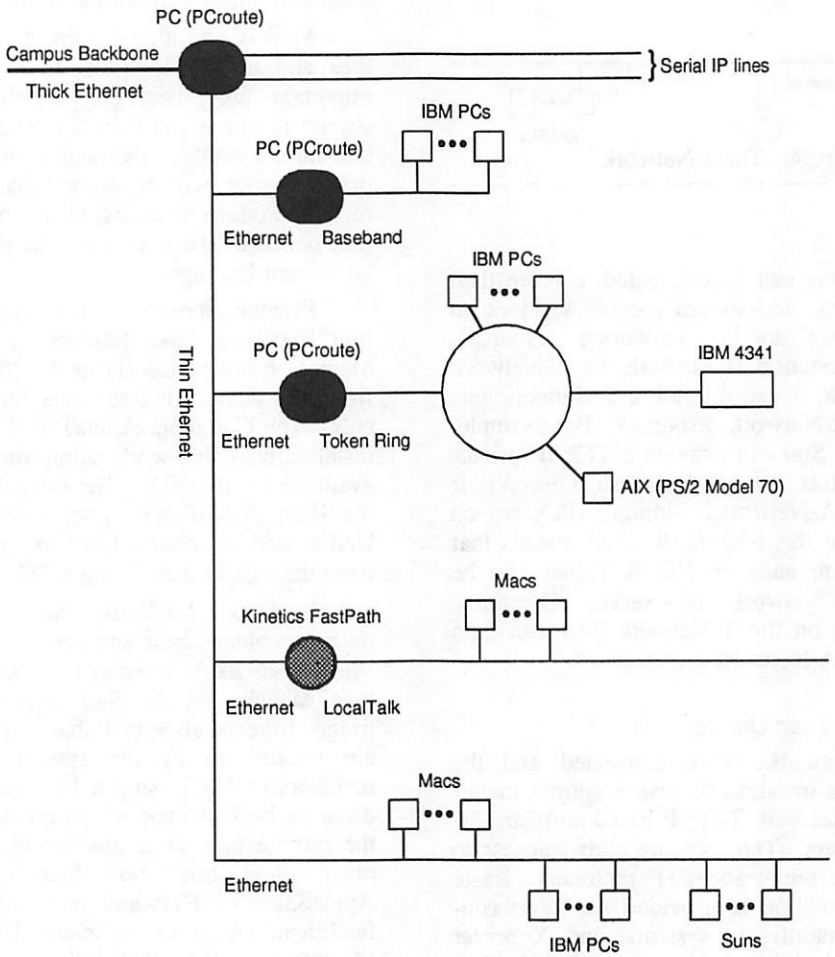


Figure 2: The T-Network

most useful clients is an lpd printer daemon for LaserWriters. The A-Network supports services such as mail, printing, and file sharing.

3.2.3 The J-Network

The J-Network connects machines communicating with the JANET protocol. A PC running the JANET server software [COWAN88c] transfers JANET packets between the Token-Ring, PC Network Baseband and Ethernet networks. All of the hosts on the J-Network are PCs. Services on the J-Network include printing, and disk sharing.

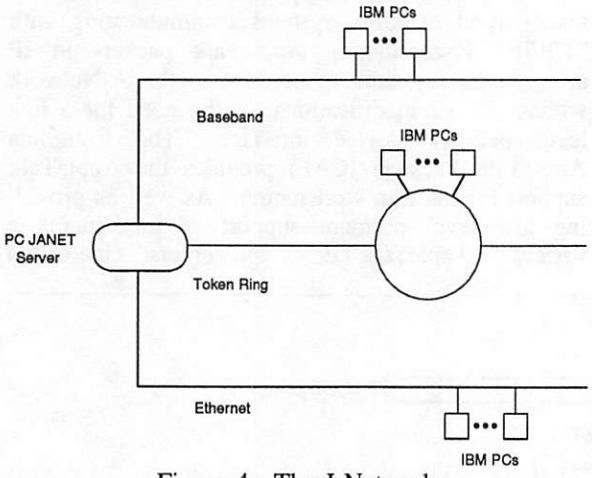


Figure 4: The J-Network

3.2.4 Synergy

Host computers can be connected to more than one logical network, and so can provide services on one network using another network's resources. Since a Sun workstation is on both the T-Network and the A-Network, it can provide a T-Network service which uses A-Network resources. For example, by using CAP the Sun can provide a TCP/IP printer service which prints on a Macintosh LaserWriter connected to the A-Network. Similarly PCs are on the T-Network and the J-Network which means that T-Network software such as NCSA Telnet can be loaded from the J-Network disk- server. Of course since the PCs are on the T-Network they can gain indirect access to A-Network services.

3.2.5 Bringing the User Online

Once the networks were connected and the appropriate drivers installed, it was a simple matter to provide each user with TCP/IP-based software for their microcomputers. The software allows access to any host using the telnet and FTP protocols. Basic 3270 terminal emulation is provided for those connecting to IBM mainframe systems, and X server software allows microcomputer users to do Motif or Open Look development.

4. UNIX Network Services

Once connectivity was achieved, it was obvious that some useful services could be provided to access common devices and streamline procedures. Minimizing a user's exposure to unfamiliar systems is also an important goal, since only a few members of CSG are directly involved with UNIX.

Electronic mail is one of the most important services. One of the Sun workstations was chosen as the central mail server. Users can of course forward their mail elsewhere, but a single machine address is used in our communication with those outside CSG.

A UNIX system was the ideal choice for mail service because of its stability and reliability. The Sun workstation can also be accessed in a number of ways - via telnet, by phone, by direct serial connection, or through a campus-wide Sytek broadband network - thus providing users access to their mail even when off-campus.

The ELM mailer is used by those who read their mail on UNIX systems, while a Post Office Protocol (POP) server allows mail to be read from microcomputers without the need to login.

Mail is considered a critical resource, so spool files and users' individual mail folders (and other important user files) are distributed nightly to a second machine and kept for several days. If a system failure disables the central mail server for more than a day or two, the second machine can be easily reconfigured to take its place and important mail files restored. This service is in addition to the regular system backups.

Printer access is also quite important. The four PostScript laser printers are accessible to any Macintosh on the LocalTalk or Ethernet network and from any device that supports the AppleTalk protocols. The Columbia AppleTalk Package (CAP) was installed on a Sun workstation, making those printers available from UNIX. The other UNIX systems and the IBM PCs simply spool their print files to the UNIX system where CAP is installed. Spooling from the PCs is done using a PC equivalent of lpr.

Backup facilities are essential in a microcomputer-based software development facility. We use an Exabyte 8mm tape drive with a 2 gigabyte capacity on the Sun workstation (a Sun cartridge drive is also available). The UNIX systems are backed up by the system administrator, but members of CSG using a PC can also use the tape drive to backup their microcomputers using tar and the remote tape protocols for PCs. Macintosh and other users can copy files up to UNIX via AppleShare or FTP and then use the Sun backup facilities. (A more efficient backup system for Macintosh systems that will copy directly to tape is currently being considered.)

The backup facility has turned out to be quite popular: each PC can be backed up with a minimum of fuss on the user's part. Previous backup schemes used the JANET server, so the tape backups free up valuable file server space. Another advantage is that users can make complete backups of their systems instead of just saving project-related files.

Filesystem sharing is not a significant issue at this point because each microcomputer has access to additional resources through the JANET and MacJANET LAN servers. File transfers and remote printing seem to satisfy most of our needs. Our UNIX systems do share disks through NFS, and an option to use the Columbia AppleTalk Package (CAP) allows Macintosh users access to UNIX files via AppleShare (an Apple-defined file server protocol).

Name service is provided by one of the Sun workstations, which in turn queries the campus name servers when necessary.

Network news is read remotely from one of the campus news servers, saving our own disk space in the process. Users must login to a UNIX system to use this facility, as we haven't felt it necessary to install and support micro-based news readers.

5. Preserving a Familiar Environment

The hardest objective to meet is that of allowing users to continue to operate in a familiar environment and insulating them from the many technical details involving the CSG and University networks and UNIX support services.

Part of this insulation is handled by designating one member of the CSG (in this case, the UNIX system administrator) to act as a network resource person, responsible for IP number assignment, network management (in coordination with the campus network group), and configuration of systems for TCP/IP access.

Insulating users from UNIX is a task that is only partially complete. Currently, support and administrative staff use electronic mail to relay phone messages and memos. A POP client program for the Macintosh allows them to do this with no UNIX contact. Similarly, printing can be done directly from a PC via `lpr` and `lpq` equivalents.

Programs from the university's *xhier* software distribution (an automatic software distribution facility described elsewhere in these proceedings), including a university-wide phone database and commands to access the tape drive, make the system easier to administer and maintain. Of course, ample documentation, printed and electronic, is available to those who need it.

6. Observations and Conclusions

Our experiences have led us to make observations and conclusions which might interest others in similar situations:

Subnets are extremely useful. CSG's internetwork is a subnet as it is almost self-contained. The network can operate with minimal external network services and can also easily be isolated from the campus network since the only connection is through a single router. Performance is more than adequate, because most of our network communication stays within the CSG network, and most communication can continue even when the campus network is down.

Our use of subnets has also had an unexpected benefit: the campus network is currently undergoing a massive reorganization to simplify its management and increase its efficiency. This reorganization has almost no impact on the CSG subnet.

Three minor problems occur when the services of the campus network are not available. Name service is lost as well as connection to the rest of the University and the national TCP/IP network. Loss of name service will be remedied soon by setting up a separate subdomain for CSG within the university's domain.

Microcomputer-based routers are flexible. Several months ago we installed a cluster of AIX-based systems on the Token-Ring network. The cluster was linked using AIX's Transparent Computing Facility (TCF) to provide a distributed environment. At about the same time, many of the older ULTRIX-based systems on campus (of which we had none) started crashing. Eventually the problem was traced to our AIX system: it was generating LARP packets that triggered a bug in the ULTRIX systems. The ideal solution would have been to upgrade the crashing systems to a later version of ULTRIX that fixed the problem, but that was not feasible. Instead, we changed our routers to filter out any LARP packets - a minor modification, but only possible because we were using a microcomputer-based router to which we had the source code.

Free software yields mixed benefits. Much of the software we use, both on UNIX and on the microcomputers, is available for free from a number of sites (see Appendix A). Because it is free, however, there are a number of risks involved in its use. You are likely to find yourself acting as a beta-test site for the software, even if you do not realize it. Since there is no obligation of support, you must be prepared to track down any bugs and fix them yourself. We had to do this with the freeware packet driver we were using on our IBM PCs, and it was a frustrating task.

A related risk is that of having the free software you use become commercial software, which is what has happened with the PCroute software we use for our routers.

Network management can always be improved. The ULTRIX-killer packets were proof of this: better tools would have allowed us to locate the problem much more quickly.

Microcomputers add administrative headaches. Nothing is static, but unfortunately system configuration files on most microcomputers cannot be updated automatically when changes occur. The network administrator has to assemble different sets of configuration files and drivers for each type of microcomputer and its corresponding network.

More integration means less security. CSG is a research organization: internal security is much looser than would perhaps exist in other organizations. External security is more of a concern, and so we are careful to run COPS and similar programs regularly on the UNIX systems and to disconnect from the campus network when network-related problems present themselves. As well, a microcomputer has to be explicitly configured in order to allow FTP access to it from another computer, and this is never done permanently.

Author Information

Peter Bumbulis is a PhD candidate in the Department of Computer Science at the University of Waterloo and a research associate with the Computer Systems Group. His current research interests are programming languages and networking. He can be contacted by mail at the Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, or electronically at peter@csg.uwaterloo.ca.

Donald Cowan is a Professor of Computer Science and a Senior Associate Scientist in the Computer Systems Group at the University of Waterloo where he received a PhD in Applied Mathematics in 1965. He is the author of over 100 publications in computer science and his current research interests include computer-communications, user interfaces and highly interactive applications, and software engineering. He can be contacted by mail at the Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, or electronically at dcowan@csg.uwaterloo.ca.

Eric Giguère is a researcher with the Computer Systems Group at the University of Waterloo where he is currently working on his MMath in Computer Science. Part of his responsibilities include system and network administration, and his research interests include user interfaces, object-oriented programming and hypertext databases. For fun he likes to write technical articles and papers. He can be contacted by mail at the Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, or via electronic mail at giguere@csg.uwaterloo.ca.

Terry Stepien is an Adjunct Assistant Professor of Computer Science and Associate Director of the Computer Systems Group at the University of Waterloo where he received an MMath in Computer Science in 1988. He has acted as team leader in a number of research projects on local area networks, user interfaces and highly interactive applications, and compilers. He can be contacted by mail at the Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, or electronically at tms@csg.uwaterloo.ca.

References

[COWAN88a] D. D. Cowan, S. Fenton, J. W. Graham, and T. M. Stepien. Networks for education at the University of Waterloo. Computer Networks and ISDN Systems, 15:313-327, 1988.

[COWAN88b] D. D. Cowan, S. Fenton, A. Pittman, and T. M. Stepien. Diversity, accessibility and adaptability - data communication needs for higher education, the University of Waterloo experience. In IEEE International Conference on Communications, pages 1576-1580. IEEE Communications Society, 1988.

[COWAN88c] D. D. Cowan, T. M. Stepien, and R. G. Veitch. A network operating system for interconnected lans with heterogeneous data-link-layers. In Proceedings of the 13th Conference on Local Computer Networks, pages 99-105. IEEE Computer Society Press, October 1988.

Appendix A: Sources

The following descriptions provide sources for many of the packages described in the paper.

PCroute is the software used on all the microcomputer-based routers for routing IP datagrams. The software was developed at Northwestern University for use with IBM PC XT- and AT-class microcomputers. These microcomputers are very inexpensive and support a large number of network adapter cards, so a complete router can be configured at minimal cost. We use it to bridge thin Ethernet to thick Ethernet, Token-Ring to thin Ethernet, and PC Network Baseband to thin Ethernet. A router can also support up to two SLIP links.

PCroute was distributed as freeware. Versions 2.1 and 2.2 can be obtained by anonymous FTP from acns.nwu.edu in the `pub/PCroute` directory. However, Northwestern University has licensed the software to LANPort, Inc., who will sell future

versions commercially. Details can be obtained by sending mail to Lanport@cup.portal.com. Existing versions of PCroute are still being distributed as freeware.

MacTCP is used to provide basic TCP/IP support on the Macintosh. It is a commercial product from Apple that is site-licensed to the University. The POP client we use for mail service on the Macintosh requires MacTCP. However, the NCSA Telnet for the Macintosh does not require MacTCP.

IBM PC packet drivers are available by anonymous FTP from sun.soe.clarkson.edu in the pub/packet-drivers directory. Commercial packet drivers are also available, most notably from FTP Software.

Telnet and FTP programs are freely available from the National Center for Supercomputing Applications (NCSA) by anonymous FTP at ftp.ncsa.uiuc.edu in the NCSA_Telnet directory. Subdirectories there contain versions for the PC (which require appropriate packet drivers) and for the Macintosh (which are self-contained). The PC version includes telnet, FTP, lpr and lpq programs.

A version of NCSA Telnet for IBM PCs with 3270 terminal support, tn3270, also exists. A version can be found for anonymous FTP on ftp.uu.net in the networking/ncsa2.2tn directory.

For Suns running SunOS 4.1.1, a version of tn3270 can also be found on ftp.uu.net as tn3270.4.1.1.tar.Z.

IBM PCs running Windows 3 can obtain a version of telnet called WinQVT from the FTP site cica.cica.indiana.edu in the file pub/pc/win3/util/qvtnet16.zip. It requires the IBM PC packet drivers mentioned earlier in this Appendix.

Columbia AppleTalk Package is available from the ftp site ftp.uu.net as networking/cap/cap60.tar.Z. It runs on a number of UNIX systems.

POP servers and clients are available from a number of locations. The POP server we use on our Sun is available from ftp.cc.berkeley.edu in the pub/popper directory. The Macintosh POP client we use is Eudora, available from ux1.cso.uiuc.edu in the mac/eudora directory. It requires the MacTCP package. An IBM PC POP client and another Macintosh client can be found on boombox.micro.umn.edu in the pub/POPmail directory.

Remote tape support is provided by the PC/TCP package from FTP Software. For information send mail to info@ftp.com. Price lists and documentation are also available for anonymous FTP at vax.ftp.com in pub/datasheets.

X Window servers for the Macintosh and the IBM PC running DOS are available from a number of commercial sources.

The Design and Implementation of a Multihub Electronic Mail Environment

Nichlos H. Cuccia - Computer Sciences Corporation

ABSTRACT

Managing an electronic mail environment on a handful of hosts from the same vendor that run the same version of the UNIX operating system is a relatively straightforward task. On the other hand, managing an electronic mail environment on several hundred machines that run different dialects of UNIX can be a cumbersome process. While distributed administration tools such as Sun's NIS are useful for centralized maintenance of files across different machines, security concerns often make their use undesirable. In addition, NIS is not available on all platforms that administrators may be required to support.

This paper describes the evolution of the electronic mail environment at NASA's Numerical Aerodynamic Simulation (NAS) facility. Vendor-supplied tools common to all supported UNIX platforms at NAS are used to centrally manage mailer configurations, mailing list files, and nameservers. The final result is an electronic mail system that is relatively easy to manage, minimizes the knowledge NAS users need to have in order to send mail, and is not susceptible to failure caused by the loss of a single hub. Pitfalls that have been encountered during the testing process will be discussed, as well as areas that are opportunities for future work.

Introduction

The Numerical Aerodynamic Simulation (NAS) facility at NASA Ames Research Center consists of over three hundred machines scattered among several local- and wide-area networks. A staff of approximately 300 government and contractor employees provide support, produce software, and perform research in areas of benefit to the 1200 industrial, educational, and government researchers that use the NAS facility.

Relatively little attention has been paid to the electronic mail environment, resulting in a number of problems. The primary problem was the lack of a redundant mail gateway or exchanger. While the NAS facility did have a machine designated as a mail exchanger, it merely forwarded mail for users that it knew about to a host where accounts for all NAS users resided. In addition, mail sent to *username@nas.nasa.gov* would eventually be bounced if the exchanger was down for a period of time more than several days.

Another problem was that anybody that wished to send mail to a NAS user needed to know the name of a NAS host on which the recipient had an account. This caused problems if the name of the host changed, or if the host was down for a substantial period of time (either for repairs or for preventive maintenance). Even if the recipient machine was up, it was an added level of complication; misspelled hostnames and mistyped user@host combinations have been among the more common sources of mailer error at our facility.

Finally, there was no single person or group responsible for electronic mail at NAS. Mail sent to the *postmaster* alias was often handled by an individual who was not versed in the management of aliases on the mail exchanger host, or who was unable to diagnose mailer problems. In addition, while the versions of the *sendmail* mailer were updated to version 5.61 after the Internet worm of November 1988, the configuration files for the mailers were not rewritten to take advantage of economies of scale. I found this somewhat surprising, since the majority of hosts at the NAS facility were either Sun or Silicon Graphics (SGI) workstations.

Goals

Because of these problems, a team of NASA and CSC personnel began work on reorganizing the NAS mail environment. We had four primary goals at the beginning of the project:

1. *Implement a redundant mail hub system.* Each mail hub would be able to route mail bound for any NAS user or alias to its destination. Furthermore, a backup machine would always be available if the primary mailhub was down for a lengthy period of time.
2. *Simplify mail addresses for NAS users.* In order to cut down user errors, we decided to eliminate hostnames from the return addresses in mail sent from NAS computers. Mail sent by NAS users to other NAS users would simply have *username* as its return address; *username@nas.nasa.gov* would be the return address for mail sent by NAS users to users

not on NAS computers.

3. *Allow users to receive mail on more than one host.* Users are able to receive mail on any (or all) hosts that they have accounts on. Those that wish to receive mail on a single host may forward their mail using a *.forward* file in their home directory on all other hosts. In addition, users are able to send mail to *postmaster* when they wish to have their default mail host changed to another host.
4. *Minimize administrative tasks.* In order to cut down on administrative headaches, we sought to reduce the number of mailer configuration files that required human maintenance to the bare minimum. In addition, we wanted to design a scheme that allowed for centralized management of mail aliases files, in order to minimize maintenance tasks on primary and backup mail hubs.
5. *Use software already in use at NAS.* Due to a lack of personnel, as well as other considerations, we were forced to limit ourselves to software already in use at NAS. Because of this, the use of mailers such as IDA Sendmail[Lo87a], the Zmailer[Zac88a], Smail[Kar90a], or MMDF-2 [Kin88a] was not an option for this project.

Design and Implementation

This section describes the framework underlying our project, as well as the design and implementation decisions that were made during the course of the project. Tradeoffs that were involved in the design process will also be discussed, as well as problems that were discovered during implementation.

General Framework

Our systems are configured as follows:

- University of California-Berkeley (UCB) *sendmail* version 5.61 on all Sun and SGI workstations and fileservers, and vendor-supplied versions of *sendmail* corresponding to at least UCB version 5.61.
  As a result of the Internet worm incident in November 1988, most UNIX vendors that use *sendmail* as their mail transfer agent are basing their version of sendmail on at least UCB version 5.61. Any vendor-supplied *sendmail* based on UCB version 5.61 or later should work fine, provided that it also supports use of the nameserver for hostname and mail exchanger (MX) resolution.
- UCB *bind* nameserver, version 4.8.3, on primary mailhub/nameserver (Sun SPARCstation I) and Sun fileservers. Any recent version of *bind* should be okay, but problems have been reported with the version of *bind* that is supplied with SunOS, especially when used in conjunction with NIS.
- Vendor-supplied mail delivery agents (*/bin/mail*).
- */usr* filesystem on Sun 680x0 filesystems remote mounted from fileservers; resident on local disks on Sun SPARC and SGI workstations and fileservers, and on Cray and Amdahl computers. This is important because the location that our *sendmail* wants its configuration file to be (*/usr/lib/sendmail.cf*) is on a remotely mounted partition that is the active */usr* partition on our fileservers. Because of this, */usr/lib/sendmail* on our Sun 680x0 workstations and fileservers is a symbolic link to */etc/sendmail.cf*, which resides on the local disks of both workstations and fileservers.
- */usr/nas* filesystem (locally-maintained software) remotely mounted on Sun and SGI workstations; resident on local disks on Sun and SGI fileservers, and on Cray and Amdahl systems.
- Sun's NIS system management tools are not in use for mail or hostname resolution purposes. This is, in part, because a number of our supported platforms do not support NIS, and have to rely on standard system files for system network configuration information.

In short, we sought to start with a system setup that does not rely heavily on custom or nonstandard software; our few system setup peculiarities arise from a need to reduce the amount of disk space on our workstations' local disks that is occupied by system software.

Mailhub Configuration

Our mailhub configuration strategy was pretty straightforward. Before this project had been conceived, our nameserver environment had been redesigned to better serve our network environment. Previously, there had been one primary nameserver that was recognized by the outside world, with another that was recognized by *nas.nasa.gov* hosts.

This had been changed to a single primary nameserver; secondary nameservers that received their updates via asynchronous zone transfers had been set up on each of our subnets in order to reduce system and network loads caused by nameserver queries. In the near future, our primary nameservice machine will be moved to an outside facility operating on an uninterruptible power supply. This will guarantee mail forwarding and nameservice operation while the main NAS computer facility is down for preventive maintenance of its cooling and power systems.

The primary nameserver is also designated to act as primary mail exchanger for the *nas.nasa.gov* domain. It is backed up by a secondary nameserver residing on our backbone subnet. This server is

backed up by the secondary nameservers on the rest of our subnets. This guarantees both internal and external mail users that there will be at least one host on our network that will be ready to handle mail sent to *username@nas.nasa.gov*.

Aliases Configuration Management

Before this project began, mail aliases management was haphazard, at best. It was not updated with new or updated information on a timely basis, and aliases for users were often missing. Most aliases on the mail exchanger simply forwarded mail to the Amdahl, which had user accounts for all NAS users.

Under the new management system, the NAS postmaster receives mail from our user database system when a new account is created. The postmaster updates a global aliases file on the primary mail exchanger, then runs a shell script that creates and installs a new */usr/lib/aliases* file on the primary hub and *rdist*s the new global file to the secondary hubs. On these machines, a script invoked by *cron* creates and installs the new */usr/lib/aliases* files. This process gets repeated once every day.

Mailing list management has also been changed, in order to reduce the amount of work required of the postmaster. In the past, the postmaster set up the mail aliases within the */usr/lib/aliases* file on the Amdahl, and was completely responsible for its upkeep. Now, the postmaster sets up *mailing-list*, *owner-mailing-list*, and *mailing-list-request* entries in the global aliases file when requested by a user, with the *mailing-list-request* and *owner-mailing-list* aliases forwarding mail to the user responsible for the alias. the *mailing-list* entry simply forwards mail to *mailing-list* at the alias owner's home machine. On this machine, an entry in the */usr/lib/aliases* file would refer to a file in the alias owner's home directory that contains the addresses in the mailing list. For example, in the case of a list residing on a user's workstation:

mailing-list:include:/u/wk/username/mailing-list

which is a file that resides in the user's home directory, and is maintained by the owner of the alias.

Mailer Configuration

Several policy issues dictated the design of the new *sendmail* configuration files, with the desire for ease of maintenance dictating the implementation of the configuration files. The policy issues were:

- Mail to NAS users should not, barring typographical errors, fail to reach a machine where the recipient reads his or her mail. This means that if the recipient *username* does not have an account on the sender machine, mail sent to *username* will be forwarded to a mailhub machine, which will then forward it to a machine where *username* does have an account.
- Addresses that worked prior to the new mail system would still work. Mail sent to *username@hostname* would reach *username* on host *hostname* if it did before the new system was put in place.
- Return addresses would have a standard format. Mail sent to NAS user *user* would have *sender-name* as its return address; mail sent to *user* at some non-NAS host would have *sender-name@nas.nasa.gov* as its return address.
- The only mail that non-hub hosts should forward to a hub is mail that it cannot deliver itself.
- Testing should be done on a subset of NAS machines. During the test period, mail sent to NAS machines not participating in the tests must have *sender-name@nas.nasa.gov* as its return address, in order to guarantee the recipient's ability to reply to the return address.

In order to better maintain our new *sendmail* configurations, we decided to use the *cf* directory out of the UCB version 5.65 distribution as a base for our rewrite of the sendmail configuration files. The UCB structure not only isolates individual rulesets into separate files, but also isolates system-dependent issues (such as UUCP mailers for hosts that need them, and variations in *Mlocal* and *Mprog* mailer flags between various platforms) in a directory specifically meant for system-dependent data. As Phil Lapsley said in his notes on the *cf* directory layout, "Trying to maintain multiple *.cf* files on

```
# resolve local domain-addressed traffic.
R$*<@$D>$*      $#local$:$1                          in local pwd or alias
R$*<@$=X.$D>$*  $#tcpld$@$2.$D$:$1@$2.$D$3     test user@host.dom
R$*<@$*.$D>$*   $#tcpld_o$@$2.$D$:$1@$2.$D$3  non-test user@host.dom
# hide behind our internet relay when talking to people in root domains
R$*<@$*.$*>$*   $#tcp$@$2.$3$:$1<@$2.$3>$4     user@non-NAS host.dom
# remaining names must be local
R$+             $#local$:$1                          in local pwd or alias
```

Figure 1: Excerpt from ruleset 0, first transitional client *sendmail.cf*

separate machines will lead to insanity."

Design

Designing the configuration file for the non-hub machines was an iterative and often painful process. Ideas that worked well during early testing failed miserably when more machines were brought into the test group. At one point, sendmail configuration files on every NAS workstation had to be replaced when an unanticipated situation caused mail to start bouncing out of control.

Our original strategy was to use following logic from Table 1.

The idea was to have the non-hub machine perform as little mail processing as possible, instead relying on the hub machine to deal with unknown mail. It was simple, and it worked–provided that users did not have a *.forward* file containing something like:

```
|/u/wk/username/lib/mh-6.7/slocal
                    -user username
```

Because this doesn't match any of the criteria being checked for, it was forwarded to the hub without any attempt at local delivery being made. Unfortunately, one of the recipients of *postmaster* mail did have such a *.forward* file. Since he was on the NAS postmaster alias, this caused a feedback loop; mail sent to *username* on his home machine would get sent to the above program, which was passed up to the hub. The hub knew nothing about such an address, because there was no such command. This resulted in a bounced message being sent to *MAILER-DAEMON*, which was forwarded back to *username* on his home machine, which starts the cycle over again.

```
# resolve local domain-addressed traffic.
R$*<@$D>$*       $#local$:$1                      in local pwd or alias
R$*<@$=X.$D>$*   $#tcpld_o$@$2.$D$:$1@$2.$D$3   non-test user@host.dom
R$*<@$*.$D>$*    $#tcpld$@$2.$D$:$1@$2.$D$3     test user@host.dom
# hide behind our internet relay when talking to people in root domains
R$*<@$*.$*>$*    $#tcp$@$2.$3$:$1<@$2.$3>$4     user@non-NAS host.dom
# remaining names must be local
R$+              $#local$:$1                      in local pwd or alias
```

Figure 2: Excerpt from ruleset 0, second transitional client *sendmail.cf*

```
if (mail bound for recipient@nas.nasa.gov) then
        rewrite address to recipient;
if (address is recipient@some NAS host) then
        forward to host via tcp-local mailer;
if (address is recipient@some non-NAS host) then
        forward to host via tcp-remote mailer;
if (recipient is known locally) then
        use local mailer for delivery;
else
        forward to hub via tcp-local mailer;
```

Table 1: Original Logic

```
if (mail bound for user@nas.nasa.gov) then
        rewrite address to user;
if (address is user@some NAS host) then
        forward to host via tcp-local mailer;
if (address is user@some non-NAS host) then
        forward to host via tcp-remote mailer;
if (user or list was known locally) then
        use local mailer for delivery;
if (user or list was known globally) then
        forward to host using tcp-local mailer;
else
        use local mailer for delivery;
```

Table 2: Modified strategy

Many megabytes of bounced mail later, a modified version of the above strategy was developed (see Table 2).

This strategy solved the forementioned problem, but highlighted another one. Many NAS users have taken to using the following *.forward* file to forward their mail to their primary mail host:

username@nas.nasa.gov

Since addresses of this form were reduced to just *username*, this was causing a mail loop within the machine. This was unfortunate, since the collapsing of *user@nas.nasa.gov* to *user* was done to avoid two unnecessary hops–one to the hub machine, the other back to the originating host.

The final strategy solves the above problem, as well as takes into account our desire to have working return addresses when mail is sent to machines not involved in the testing process (see Table 3).

During the testing period, the test for *user@some NAS host* was split into two tests. One test determined whether the host was part of the test group, in which case it used the *tcp-local* mailer for forwarding purposes. If the host was not part of the test group, then the *tcp-local-orig* mailer was used.

The design of the mail hub's configuration, by comparison, was much more straightforward. It uses the strategy in Table 4.

During the testing period, the strategy for delivering mail to *user@some NAS host* was amended in the same way as the final non-hub strategy.

Implementation

This section of the documentation contains implementation details of the NAS sendmail code. Readers that are not familiar with *sendmail* configuration files should refer to [All86a] or [Nem89a]

Mailer Mapping

Implementation of the mailhub strategy described above is relatively straightforward. The code in ruleset zero that implements it is shown in Figure 1. In Figure 1, as in all excerpts of *sendmail.cf* code, the `$D` macro is defined as the local *domain* name. The *X* class referred to in the second rule from the top contains the names of all machines that are running the new version of the *sendmail.cf* file; all other local hosts are still running with their original configurations.

As more machines are tested, the *X* class will contain the names of all machines *not* using the new configuration file. The last few lines in ruleset zero will also need to be changed; these changes are shown in Figure 2. When the testing is finished, the test for hosts in the test class will be removed, as will the X class itself.

The strategy used by the non-hub configuration file is a little more difficult to implement. The implementation is shown in Figure 3.

```
if (address is user@nas.nasa.gov) then
        forward to hub via tcp-local mailer;
if (address is user@some NAS host) then
        forward to host via tcp-local mailer;
if (address is user@some non-NAS host) then
        forward to host via tcp-remote mailer;
if (user or list was known locally) then
        use local mailer for delivery;
if (user or list was known globally) then
        forward to host using tcp-local mailer;
else
        use local mailer for delivery;
```

Table 3: Final Strategy

```
if (address is user@nas.nasa.gov) then
        use local mailer for delivery;
if (address is user@some NAS host) then
        forward to host via tcp-local mailer;
if (address is user@some non-NAS host) then
        forward to host via tcp-remote mailer;
else
        use local mailer for delivery;
```

Table 4: Mail hub configuration design

This piece of code invokes uses three additional classes, each of which refers to an outside file. The *A* class contains the names of all aliases that exist on the local machine. It is declared in the configuration file as follows:

```
FA/etc/aliases.lst
```

The *L* class contains the names of all logins that exist on the local machine, and is declared in the configuration file as follows:

```
FL/etc/passwd.lst
```

The *G* class contains the names of all logins and aliases that are known by the hubs. It is declared in the configuration file as follows:

```
FG/usr/nas/lib/aliases.global.lst
```

Note that in the case of workstations, the */usr/nas* filesystem is a remote filesystem, exported by the fileservers. Since the fileservers double as backup mail hubs, the work involved in maintaining the file across all NAS hosts is decreased significantly.

A *perl* script invoked by *cron* creates the */etc/aliases.lst* and */etc/passwd.lst* on the workstations, and */usr/nas/lib/aliases.global.lst* file on the fileservers. Because *sendmail* does not dynamically update file-based classes, the script also kills and restarts the *sendmail* daemon, which is required in order for changes to take effect.

The *D* variable is defined as the domain *nas.nasa.gov*. With the exception of the second and third rules, it is used in every case that a *tcpld* or *tcpld_o* mailer is used. Through this mechanism, mail that can't be resolved by the local machine will be handled by the first hub that can answer the nameserver query and process the mail.

Return Address Rewriting

Return address rewriting is done in rulesets 17 (for mail to other NAS hosts) and 14 (for mail to non-NAS hosts), and is relatively straightforward. Figures four and five illustrate the address rewrites.

The one subtlety is the introduction of the `S` class. This class contains the names of accounts that should not have the originating hostname stripped from the return address. Addresses in this class include *root*, *postmaster*, and *mailer-daemon*. This allows systems administrators and postmasters to readily determine where a bounced message, *cron* error message, or other indicators of trouble are originating from without having to wade through the header of the message.

Future Improvements

This document describes the first, and most dramatic, phase of the implementation of the NAS mail environment. The system has been in place since mid-June, and the response from users and administrators has been encouraging. Many of our users are not very skilled in the use of electronic mail, and greatly appreciate being able to send mail to another NAS user without having to know any information other than their login name. Administration has proven to be relatively straightforward; maintenance of the *sendmail.cf* files has been greatly simplified, due to the use of a common base for all platforms. *aliases* management has also been simplified; the postmaster now updates one of two files and executes an *rdist*, which distributes the file and performs local modifications where necessary. Finally, a brief period where arp table limits on the primary hub was resulting in its crashing less than a minute after it last rebooted resulted in no interruption in the NAS user's ability to communicate via electronic mail with NAS and non-NAS recipients.

While significant improvements have been made with respect to our original mail environment, the current system has a number of glaring weaknesses:

- If a user wishes to have their mail host changed from one host to another, they need to send mail to *postmaster*. This makes more work for the person who is responsible for dealing with *postmaster* mail, and may not become functional on NAS machines until the next business day.
- The use of static files to define classes that

```
# resolve local domain-addressed traffic.  R$*<@$D>$*
$#tcpld$@$D$:$1@$D$2          user@domain R$*<@$=X.$D>$*
$#tcpld$@$2.$D$:$1@$2.$D$3    nontest user@host.domain R$*<@$*.$D>$*
$#tcpld_o$@$2.$D$:$1@$2.$D$3  test user@host.domain # hide behind our
internet relay when talking to people in root domains R$*<@$*.$*>$*
$#tcp$@$2.$3$:$1<@$2.$3>$4    user@host.domain R$=A
$#local$:$1                   in local aliases R$=L
$#local$:$1                   in local passwd R$=G
$#tcpld$@$D$:$1@$D            global file R$+
$#local$:$1                   everybody else.
```

Figure 3: Excerpt from ruleset 0, hub *sendmail.cf*

determine the disposition of mail is not an optimal solution because it requires the restarting of the *sendmail* daemon every time the files are updated. Worse, the information in */etc/passwd.lst*, */etc/aliases.lst*, and */usr/nas/lib/aliases.global.lst* is redundant, since it duplicates information that already exists in */etc/passwd*, */usr/lib/aliases*, and */usr/nas/lib/aliases.global*. This problem is partially resolved by *sendmail* version 5.65, which, when compiled with the SCANF preprocessor directive, allows a *scanf*(3) format string as a second argument to file class definitions. For example, the previous definition of the `L` class would be rewritten as:

```
FL /etc/passwd %[^:]%*[^\n]\n
```

Since many sites have comments and empty lines in their *aliases* files, some preprocessing of the *aliases* would be needed. Also, since file updates are not handled dynamically, a restart of the *sendmail* daemon would still be needed.

- Users can easily set up mailer loops between machines through the use of *.forward* files. This can be done either by having *.forward* files on two machines that forward mail to the other machine, or by having a *.forward* file on the machine the hub forwards mail to that reads:

```
user@nas.nasa.gov
```

This is a result of our desire to allow users to receive mail on as many hosts as they wish, and would be less likely if users were required to receive mail at only one host.

- The sender must still know the login name of the person they wish to send mail to; users would like to be able to send mail to *firstname_lastname* where possible. A filter that takes *passwd* gecos entries and maps them into login names:

```
firstname_lastname: login_name
```

would take care of this for the sender. Without a modified version of *sendmail*, however, the return address would still be the sender's login name.

Conclusions

At the beginning of this project, we wanted to design and implement a mail environment that would process mail bound for NAS while the main computer rooms were down for periodic maintenance, simplify the addresses NAS users use when sending mail to each other, yet not rely on tools that were not already in use at NAS. Although there are shortcomings in our current implementation, we feel that we have been successful in what we have set out to accomplish.

Acknowledgements

Many people were involved in the design and implementation of the NAS mail system. The original architecture was developed by Michele Crabb, Nick Cuccia, and Craig Manning of CSC, and Louise Kokinakis and Karen Castagnera of NASA. Nick developed the *sendmail* configuration files, and Michele designed the aliases distribution scheme, while Louise managed the project and kept the NASA management appraised of our progress. Peter Cooke and Harry Waddell of CSC and John Lekashman of NASA offered valuable feedback during the design and implementation phases of the project.

The author would also like to thank Paul Ticknor, Steve Storm, and Dick Gunderson of CSC for their encouragement, and Ana Grady of NASA for allocating resources during the course of this project. Paul and Steve proofread early drafts of this paper.

```
R$~S            $:$1                user w/o host
R$=S            $@$1<@$w>           priv'd sender
R$~S<@$w>       $:$1                this host
R$~S<@$=w>      $:$1                or an alias
R$~S<@$->       $:$1                if nameserver fails
```

Figure 4: Return Address Rewriting, NAS-bound Mail

```
R$-             @$1<@$D>            user w/o host
R$+<@$w>        @$1<@$D>            this host
R$+<@$=w>       :$1<@$D>            or an alias
R$+<@$->        :$1<@$[$2$]>        canonicalize into dom
R$+<@$->        :$1<@$D>            if nameserver fails
R$+<@$=N.$D>    @$1<@$D>            nic-reg hosts are ok
R$+<@$*.$D>     @$1%$D<@$D>         else -> u%h@gateway
```

Figure 5: Return Address Rewriting, non-NAS-bound Mail

Author Information

Nichlos H. Cuccia is Network Administrator for NASA's Numerical Aerodynamic Simulation (NAS) facility at Ames Research Center. His primary areas of interest include electronic mail and messaging systems, distributed file management, and network monitoring and management. He can be reached via US Mail at NASA Ames Research Center; Mailstop N258-6; Moffett Field, CA 94035, or via email at *cuccia@nas.nasa.gov*.

This work is supported by contract NAS2-12961 of the National Aeronautics and Space Administration.

References

Lo87a. Lovstrand, Lennart, *Electronic Mail Addressing in Theory and Practice with the IDA Sendmail Enhancement Kit,* Department of Computer and Information Science, University of Linkoping, Linkoping, Sweden, 1987.

Zac88a. Zachariassen, Rayan S., *The ZMailer Operations Guide,* Artificial Intelligence Group, Department of Computer Science, University of Toronto, Toronto, 1988.

Kar90a. Karr, Ronald S. and Landon Curt Noll, *Smail-Installation and Administration Guide,* Amdahl Corporation, Sunnyvale, 1990.

Kin88a. Kingston, Douglas P., Steve Kille, Julian Onions, and Daniel B. Long, *Installing and Operating MMDF II,* Ballistic Research Laboratory, Aberdeen Proving Ground, MD, 1988.

All86a. Allman, Eric, "Sendmail Installation and Operation Guide," in *UNIX System Manager's Manual-4.3 Berkeley Software Distribution,* Computer Systems Research Group, University of California, Berkeley, 1986.

Nem89a. Nemeth, Evi, Garth Snyder, and Scott Seebass, *UNIX System Administration Handbook,* pp. 302-333, Prentice Hall, Englewood Cliffs, NJ, 1989.

A sendmail.cf Scheme for a Large Network

Tina M. Darmohray - Lawrence Livermore National Laboratory

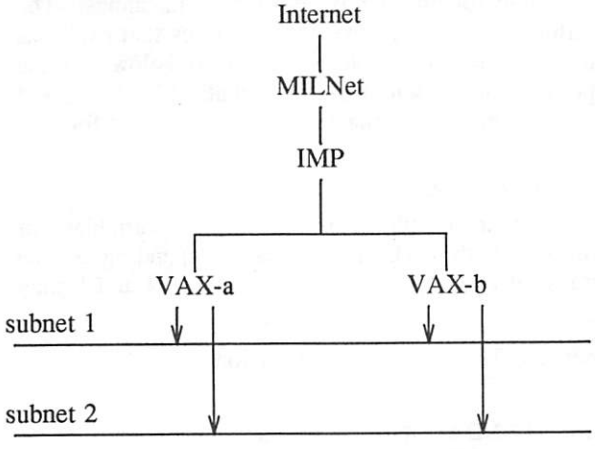
ABSTRACT

Like most large networked sites our users depend heavily on the electronic mail system for both internal and off-site communications. Unfortunately the sendmail.cf file, which is used to control the behavior of the sendmail program, is somewhat cryptic and difficult to decipher for the neophyte. So, on one hand you have a highly visible, frequently used utility, and on the other hand a not-so-easily acquired system administration forte. Here is the sendmail topology of our site, what premises we based it on, and the parts of the sendmail.cf files which support the topology.

Topology of Our Network in 1986

In 1986 I joined a group that needed system administration for a network of about 100 Sun Microcomputers. At that time the internal network consisted of two subnets that were hooked together by redundant VAX 11/750s that functioned as gateways. The 750s ran the subnetting code for the internal network and provided the connections to the outside world via an IMP to the MILNet.

The over-riding design goal behind the network topology that existed in 1986 was "communication." Each workstation should be able to communicate equally with every other machine on the internal network as well as any machine on the Internet:



Changes by 1989

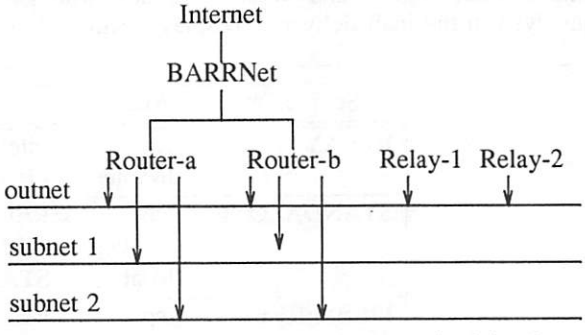
In the Fall of 1989 the Internet was hit by a worm. Our site was one of the many sites affected. No real damage was done by the worm, but it certainly heightened security awareness. The open communication model that our site used was no longer popular. More and more large sites were designing network topologies with security in mind. We decided that we needed to do the same.

Concurrently, NSFNet and Regional networks had gained in popularity as preferred routes to the Internet. We decided to pursue a BARRNet connection and retire our connection to the IMP.

Now there were several reasons to consider redesigning our Inter-network gateway connections. The VAX 11/750s were old, our Internet connection was outdated, and our security was lacking. We decided to create a new security-conscious network topology. Unfortunately, by the time this was decided, the site networking "guru" had moved on. We were left with the task ahead of us and no one who knew the particulars of how to do it. We decided to try it ourselves.

New Network Topology

We had three primary goals in designing our current network topology: we wanted to maintain unrestricted communications within the local network, continue to provide Internet access for the users, and, this time, we wanted whatever security we could provide against random hackers from the Internet:



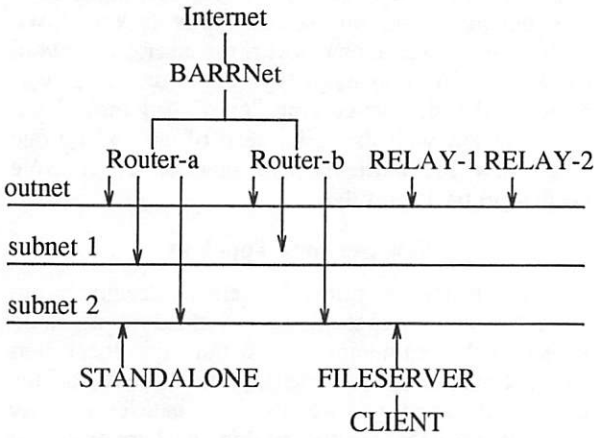
The new network topology required hardware and software changes. The routers restrict TCP traffic from passing directly from the internal network to the Internet. Since this is a single path into and out of our network we decided that we could not allow it to be a single point of failure. For that reason we purchased redundant Cisco AGSs for routers

and Sparcstation IPCs for ftp, telnet, and mail relay machines.

Network Topology According to Mail

The sendmail.cf file required changes because of the new network topology. Internet bound mail must now pass through the relay machines rather than being delivered directly by each internal machine. We also chose to rewrite the headers on all outbound mail to user@our.domain. For security, this "hides" our internal topology. It also gives us flexibility to change the hostnames of the relay machines without requiring our users to relearn or redistribute their email address; it is always user@our.domain.

We classified our network topology for mail into four classes of machines. RELAY machines are those capable of relaying mail to the Internet on behalf of an internal-network machine that does not have direct internet access. FILESERVER machines contain /usr/spool/mail for diskless clients. STANDALONE contains its own /usr/spool/mail (as does a FILESERVER). CLIENT mounts its mail from a FILESERVER:



Source machine and destination address determines what mailer and what machines will get involved in the mail delivery. Display 1 shows how these combinations are handled.

Implementing the New Mail Topology

There are two files which implement the new mail scheme. The Makefile generates sendmail.cf files based on mail machine type. The sendmail.cf file itself controls the behavior of sendmail on each type of machine in accord with the new mail topology.

Make It Easy

The makefile contains rules for the different machine types. The RELAY sendmail.cf files are compiled with a special GATEWAY flag that will trigger alternative paths in the configuration file to account for their internet-bound traffic responsibilities. Each FILESERVER and STANDALONE sendmail.cf file is compiled with a flag to indicate which RELAY machine to use for Internet-bound mail. In this way, if your site can afford more than one RELAY machine, you can take advantage of load balancing and can also avoid a single point of failure. CLIENT sendmail.cf files have a FILESERVER flag defined so that local mail will be delivered to the FILESERVER, rather than to the CLIENT itself. Display 2 shows the configuration file makefile.

The sendmail.cf File

We started with a generic Berkeley sendmail.cf file. Most of the body of the sendmail.cf files are the same for all four types of mail machines. The differences relate to the different roles that each has in the sendmail topology. Included below are the portions of the sendmail.cf file that will differ based on the type of mail machine it was compiled for.

Using Variables

Sendmail allows you to define variables, or macros, with a D. In this case, depending on the flags set at compile time, the variables R and F may

Source On	Mailer	Delivered To	Destination
RELAY	tcp	Internet Host	Internet
	localtcp	FILESERVER/STANDALONE	Localnet
STANDALONE	tcp	RELAY	Internet
	localtcp	FILESERVER/STANDALONE	Localnet
	local	STANDALONE	Local
FILESERVER	tcp	RELAY	Internet
	localtcp	FILESERVER/STANDALONE	Localnet
	local	FILESERVER	Local
CLIENT	tcp	RELAY	Internet
	nfs	FILESERVER/STANDALONE	Localnet

Display 1: How destinations are handled

be set to the chosen RELAY machine and FILESERVER respectively. Later in the sendmail.cf file $variable (e.g. $R) will reference the value of the variable.

```
#if defined(RELAY)
DR/**/RELAY
#endif
#if defined(FILESERVER)
DF/**/FILESERVER
#endif
```

Sendmail also has a conditional statement that can check if a variable has been set and then act accordingly. The following statment will interpolate text1 if the macro $X is set, and text2 otherwise.

```
$?Xtext1$|text2$.
```

Here is the first instance where the variable F is checked. First j is defined as the value of w, which is defined by sendmail as the hostname of this site. Next J is defined as the value of F (a FILESEVER name) if it is defined, otherwise it will become $w, the machine name as set in the previous line.

```
Dj$w
DJ$?F$F$|$j$.
```

Classes are like set or arrays of 'words'. You define a class with the C command. Then you can test to see if something matches any value in the class with a $=class command. In the sendmail.cf file you can explicitly define each member in the class, or you can get that input from a file. We define a class of local machines, L, using the hosts.equiv file from our site.

```
#! /bin/make -f
#
#    Create configuration files for our.domain hosts
# GATEWAY is a machine that has direct internet access.
# FILESERVER names an NFS fileserver that contains "/usr/spool/mail" for CLIENTS.
# RELAY names a machine that will relay mail to the internet on behalf of
#     STANDALONE, FILESERVER, or CLIENT.
SRC     =       sendmailcf.c
CPP     =       cc -E $(SRC)
all:  relay1.cf relay2.cf
    standalone.cf
    fileserverA.cf fileserverB.cf fileserverC.cf
    client_of_A.cf client_of_B.cf client_of_C.cf
#   RELAY MACHINES
    relay1.cf relay2.cf:    S1.c
    $(CPP) -DGATEWAY > $@
#   STANDALONE MACHINES
    standalone.cf:          S1.c
    $(CPP) -DRELAY=relay1.our.domain > $@
#   FILESERVER MACHINES
    fileserverA.cf: S1.c
    $(CPP) -DRELAY=relay1.our.domain > $@
    fileserverB.cf fileserverC.cf:  S1.c
    $(CPP) -DRELAY=relay2.our.domain > $@
#   CLIENT MACHINES
    client_of_A.cf:         S1.c
    $(CPP) -DFILESERVER=fileserverA.our.domain -DRELAY=relay1.our.domain > $@
    client_of_B.cf:         S1.c
    $(CPP) -DFILESERVER=fileserverB.our.domain -DRELAY=relay2.our.domain > $@
    client_of_C.cf:         S1.c
    $(CPP) -DFILESERVER=fileserverC.our.domain -DRELAY=relay2.our.domain > $@
```

Display 2: Configuration file makefile

```
FL/etc/hosts.equiv %[^.]%*s
```

The domain name for the site is defined as D.

```
DDour.domain
```

Rulesets and Mailers

Mail addresses traverse a path through the the sendmail.cf file. The paths are transformation commands grouped into rulesets. Rulesets 0, 1, 2, 3, and 4 are somewhat constant throughout all sendmail files which have their origin in the original sendmail.cf file from Berkeley.

Mailers are defined like rulesets, but use an M rather than an S. See Display 3 for the path of addresses through a sendmail.cf file and the purpose of each ruleset.

Ruleset 0

Ruleset 0 is where an address is resolved to mailer, host, user. This means that the address has been parsed, an appropriate mailer has been selected, and the host and user information is passed to the mailer. The flags that were defined in the Makefile are used in this ruleset. The RELAY machines with Internet access can accept mail addressed to user@our.domain. So, if the machine is a RELAY machine it would choose the local mailer (a /usr/lib/aliases file is kept on the RELAY machines), otherwise the mail is shipped to the RELAY machines via the localtcp mailer for delivery. See Display 4 for an example.

```
The basic ruleset paths are:

                    ---> S0 --> resolved address
                   /
                  /       --> S1 --> MS --> S4 --> rewritten sender
                 /       /
address --> S3 ----> D --
                         \
                          --> S2 --> MR --> S4 --> rewritten recipient

Where S# is ruleset number #, D is the addition of a domain to the
sender name, MS and MR are mailer-specific sender and recipient
rewriting rules (respectively).

Within this configuration file, the following rulesets are used:
  S0    resolve address to (mailer,host,user)
  S1    sender field pre-rewriting
  S2    recipient field pre-rewriting
  S3    name canonicalization
  S4    final output post-rewriting

  S11   sender rewriting for nfs mailer
  S20   leave local recipient alone

  S14   sender for localtcp, recipient rewriting for localtcp, tcp mailers
  S15   sender rewriting for tcp mailer
```

Display 3: Ruleset layout

```
/*
 * Ruleset 0: resolve (mailer,host,user)
 */
S0
/* std. Ruleset 0 stuff - not shown */
#if defined(GATEWAY)
R$*<@$D>$*    $#local$:$1           user@our.domain
#else
R$*<@$D>$*    $#localtcp$@$R$:$1    user@our.domain
#endif
```

Display 4: Example piece of ruleset 0

Any mail initiated within the site bound for the site is also sent via the localtcp mailer. It does not have to be sent through the RELAY machines because the local machine name that it is bound for is specified. We want to keep internal network traffic localized, if possible, and send traffic to the RELAY machines only when necessary. Display 5 illustrates this rule.

Mail bound for the Internet is sent via the tcp mailer. On the RELAY machines it is sent to the Internet host designated by $@host in ruleset 0. On any other machine (that doesn't have direct Internet access) it is delivered to the RELAY host, which will forward the mail onto the Internet. See Display 6 for the rules.

If the address is local and a FILESERVER has been defined (mail initiated from a CLIENT machine), the NFS Mailer is chosen otherwise the mail is delivered locally via the local mailer. In general, we encourage mail to be delivered to a FILESERVER rather than a CLIENT or a STANDALONE machine. FILESERVERS are more robust machines, and therefore a better destination for mail communications. Display 7 shows the rule for local names.

Mailers

The NFS mailer is used to send mail to the machine which holds /usr/spool/mail since only that machine's root can write to these mailboxes. This mailer is the first example of how mailers can use mailer-specific rulesets to clean up mail addresses.

```
R$*<@$-.$D>$*       $#localtcp$@$2.$D$:$1<@$2.$D>$3     user@ourhost.our.domain
```

Display 5: Resolving to the localtcp mailer

```
#if defined(GATEWAY)
R$*<@$+>$*  $#tcp$@$2$:$1<@$2>$3  user@tcphost
#else
R$*<@$+>$*  $#tcp$@$R$:$1<@$2>$3  user@tcphost
#endif
```

Display 6: Mail bound for the Internet via the tcp mailer

```
/* everything else must be a local name */
R$+   $?F$#nfs$@$F$:$1$|$#local$:$1$.    local names
```

Display 7: Local names

```
Mnfs, P=[IPC], F=msDFMuXLI, S=11, R=20, A=IPC $h, E=\r\n

S20

S11
R$-         $@$1@$J
```

Display 8: Local and NFS Mailers

```
Mlocaltcp,      P=[IPC], F=msDFMueXL, S=14, R=14, A=IPC $h, E=\r\n
Mtcp,           P=[IPC], F=msDFMueXL, S=15, R=14, A=IPC $h, E=\r\n

S14
R$-             $1@$J                   name => name@localhost
R<@$+>$*        $@<@$1>$2               resolve <route-addr>
R$*<@$+.UUCP>   $:$2!$1<@$J>            put local name on UUCP
R$*:$*          $1.$2                   map colons to dots

S15
R$-             $@$?D$1@$D$|$1@$J$.     local sender
R$*             $:$>14$1                normal rewriting
R$*<@$=L>       $:$1@$D                 x@ourhost => x@our.domain
R$*<@$=L.$D>    $:$1@$D                 x@ourhost.our.domain => x@our.domain
```

Display 9: Internet (TCP) mailer

Notice that the NFS mailer rewrites the sender addresses with ruleset 11 so that the mail appears to have come from a FILESERVER rather than a CLIENT machine. This is done with the S=11. For example, mail initiated from user@client_of_A would be rewritten to user@fileserverA.our.domain. This is strictly for aesthetics, but we have found that it reinforces, for the users, where their mail is really being stored. This way if they change internal fileservers they can request that their mail be moved as well if they wish. Additionally, it serves as an initial pointer for the network administration team as to which fileserver, subnet, and human-group a user is affiliated with. See Display 8.

The TCP mailer is selected for Internet-bound mail. Once again, mailer-specific rulesets are used for last minute address cleanup. In this case all local hostnames are "hidden" by rewriting the sender address with ruleset 15 so that the mail appears to have come from user@our.domain rather than user@ourmachine.our.domain. This is done by testing to see if the machine name matches any of the local hostnames previously defined in the class L. If it does, then the address is rewritten. The last two lines of ruleset 15 accomplish this. See Display 9.

Conclusion

Our site has been using this new network and mail topology for about a year now. Our experience has been that it was well worth the time and money it took to implement it. The redundant relay machines and rewritten internet bound mail addresses increased our network flexibility. If one of the relay machines is down incoming mail goes to the other machine. If the relay machine will be down for a long period of time, outbound mail can easily be redirected to use the relay machine that is up. Additionally, since inbound mail is sent to our (static) domain name, the relay machines can be swapped out without any consequence to our internet mail connectivity.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.

Author Information

Tina Darmohray is the Technical Lead for the UNIX Support Team at Lawrence Livermore National Laboratory. She received her BS and MS degrees from the University of California, Berkeley. Reach her via U.S. Mail at Lawrence Livermore National Laboratory; P.O. Box 808 L-270; Livermore, CA 94550. Reach her electronically at tmd@s1.gov.

References

Allman, Eric. Sendmail - An Internetwork Mail Router. *UNIX Programmers Manual.* 4.3 Berkeley Software Distribution. University of California, Berkeley. April 1986.

Nemeth, Evi, Garth Snyder, and Scott Seebass. *Unix System Administration Handbook.* Prentice Hall, Inc. 1989.

SHARE II - A User Administration and Resource Control System for UNIX

Andrew Bettison, Andrew Gollan, Chris Maltby, Neil Russell
- Softway Pty Ltd

ABSTRACT

The UNIX operating system has become the vehicle for the current 'open system' standards movement because of its widespread availability on a large range of hardware platforms. The portability of UNIX stems from the elegant simplicity of its original design, and the sparsenesss of its original source code. Ironically, simplicity is now the main aspect of UNIX that standards bodies are striving to change, as extra functionality is grafted on to meet the diverse requirements of the marketplace.

The effort to standardise system administration facilities for large-scale, networked systems is a typical example of such an endeavour. UNIX system administration currently involves a bewildering variety of methods, so standards bodies are encouraging the development of object-oriented interfaces to achieve uniform control. There is, however, a risk that such designs may run afoul of the lack of sufficiently complete and manageable objects.

This paper discusses existing resource control problems, then describes SHARE II, its application to those problems, and its user administration features. The SHARE II system addresses shortcomings in the area of user administration by introducing a new scheme for measuring and controlling the resource consumptions and the privileges of all users. The authors believe that SHARE II has the potential to play a valuable role in future system administration developments.

Introduction

Operating systems exist to make the resources of a machine available to users in a convenient form. On multi-user systems, users share the same resources, with the benefit that the resources are better utilised. Inevitably, however, there is contention between users, especially on larger systems, giving rise to a host of administrative problems that are absent on single-user systems. The original authors of UNIX were not concerned with such administrative problems, and many system designers have since added *ad hoc* features to cope with them.

The difficulties encountered by administrators of single (non-networked) computer systems are intensified by the presence of networking. One of the shortcomings of the present approach to resource management and control in distributed environments is the inadequacy and complexity of existing basic management *objects* (such as users, quotas, devices etc.) for modeling administrative tasks.

This paper discusses the problems outlined above and examines the currently available methods for dealing with them on UNIX. It then describes SHARE II, a more complete solution. The relevance of SHARE II to distributed system management is the provision of a new, coherent *user* object. Our focus on user management presumes that the equally problematic areas of backup/restore, crash recovery, capacity planning etc. are separately solvable.

A critique of existing UNIX resource control methods

This section is a brief outline of the methods available for resource control on existing System V and Berkeley systems. We expect that the reader is familiar with most (if not all) of these, so descriptions are kept brief.

Disk Space

Users tend to be lazy when it comes to removing or archiving unwanted files. If unrestrained, users will quickly consume all available disk space. Here is a survey of the methods currently available to system administrators for controlling users' disk usage.

Usage Monitoring by Hand

When the amount of free disk space starts to dwindle, the system administrator can contact the users responsible for the largest recent increases as reported by a regularly run survey program. The feeling that they are being watched will cause users to be more reasonable.

This will only work in a small community of users in which the users are more likely to feel that the welfare of the group is important. On larger

systems, the system administrator's job becomes a tiresome duty, and automating the system simply breeds indifference.

File System Partitioning

UNIX allows disks to be arbitrarily divided into many *partitions*, each containing a file system. This was originally intended as an administrative convenience to make the configuration of a system's file systems largely independent of the available hardware. By confining groups of users to their own partitions, it is possible to prevent the disk consumption of each group from interfering with others. This scheme relieves the system administrator of the need for constant surveillance, and forces the responsibility for moderation to within each group.

When a group genuinely requires more disk space, the administrator must repartition the disks, preserving the contents of each partition. This is an intrusive and cumbersome process. This scheme also has the undesirable side-effect of fragmenting free disk space.

ulimit and RLIMIT_FSIZE

The `ulimit(2, ...)` and `setrlimit(RLIMIT_FSIZE, ...)` system calls, originating respectively from the System V and Berkeley systems, limit the maximum file seek location at which a process may write data.

It is easy to circumvent a file size bound by splitting large files into several small files, so these calls are no use for controlling long-term disk usage. Sparse files with large logical lengths become impractical. A file size bound is really only effective for limiting the size of temporary files, as the application programs that create them typically encounter the bound and fail, often without explaination, and without allowing the user to split the file. This is not very useful to system administrators, as there is usually plenty of temporary disk space available.

Berkeley Quotas

The disk quota system comes close to providing the level of control required by system administrators[1]. The main limitation of quotas is that the hard and soft disk limits apply only to a single partition - it is impossible to apply a single limit across a group of partitions.

The disk quotas on some systems exhibit the following bug: when a file changes ownership as a consequence of *chown*, the disk usages of the original and new owners are not always adjusted accordingly. It is also possible for users to defeat their assigned disk quotas by discovering files owned by another user that are publicly readable and writable.

Memory

Berkeley introduced a number of per-process limits on the use of system virtual and real memory. This was a major improvement over the pre-existing and System V kernel tunable `MAXUMEM`, which enforced a system-wide process virtual size limit.

Any per-process memory limiting mechanism has severe shortcomings when used to limit users, because of the ease with which UNIX allows process creation combined with the wide variety of methods for inter-process data exchange. The principal use of the virtual and resident memory controls, `RLIMIT_DATA`, `RLIMIT_STACK` and `RLIMIT_RSS`, has been by batch queueing systems to provide differentiation between job queues.

The `RLIMIT_CORE` limit is very handy for systems with a shortage of disk space as it prevents 'coredump' files becoming larger than the given size. As a resource control its value is limited.

CPU

The CPU is often the most contentious resource, even though many machines are, in fact, disk-bound. Monopolising the CPU is a sure way to immediately deny the machine to all other users, because the rate of use of CPU effectively determines a user's rate of use of all other machine resources.

RLIMIT_CPU

The `setrlimit(RLIMIT_CPU, ...)` system call causes the process to be killed when it has consumed a specified number of CPU seconds. This can be used to discourage users from running long, cpu-intensive processes, but can be circumvented by regularly saving the process state in a checkpoint file and restarting. In any case, there is nothing inherently bad about users running long processes, as long as their *rate* of service can be controlled.

Scheduler

The standard UNIX process scheduler is a mystery to most users. It attempts to apportion CPU cycles evenly among the competing processes, with some sort of bias selectable by means of *nice*.

Like many of the original pieces of UNIX, the scheduler makes a broad assumption that overloaded machines do not exist, and its task is therefore to simply provide regular timeslices on demand to processes. By failing to cope with the overload case, the scheduler provides incentive to users on busy machines to increase their demands, yielding a larger portion of a diminishing pie.

[1] Quotas and SHARE II have a common ancestry.

Nice and Auto-nice

The *nice* mechanism provides the scheduler with a hint that the process is less or more important than others. The interaction of *nice* with varying system busy-ness (*loadav*) is not defined. Furthermore, there is no clear relationship between a process's *nice* value and its rate of execution, even on a predictably loaded machine.

Some systems have a built-in feature that increases the nice level of processes once they have consumed a certain amount of CPU time. This fails to differentiate between long running intensive interactive tasks[2] and genuine background CPU bound tasks.

Processes

The size of the process table is determined by the `NPROC` system configuration parameter, and is not an inherent machine limitation. Controlling the number of processes that a user may have running at one time is necessary to prevent the user from spawning so many processes that the system hangs. The `MAXUPROC` system configuration parameter determines the maximum number of processes that any user may have.

Changing either of these parameters involves rebuilding the kernel; typically a time-consuming task, as precautionary backups are often required. Also, these parameters do not allow different limits for different users.

Terminals

In keeping with the concept of the super-user as a single source of all privilege, the only control offered on location of access to a UNIX system is for the super-user account. In System V usually only the console can be used, though in Berkeley any device identified as secure in `/etc/ttys` is sufficient for super-user access.

Other users are allowed access anywhere, given that they can supply a valid password. Most System V implementations allow an undocumented control for dial-up connections which requires that an additional password be specified dependent on which login *shell* the user is going to use.

Other Hardware Resources

None of the common UNIX versions provide any other access control for devices such as printers, tape drives, and so on, except by conventional file-system permissions on the device files in `/dev`. In the absence of access control lists this is allows only crude all-or-nothing control.

Some print spoolers provide a form of access control by allowing administrators to tinker with spooling and despooling scripts. The resultant controls are patchy by nature, and do not typically provide facilities for users to enquire about their status.

Accounting

An often overlooked requirement for effective system management is detailed accounting information. The administrator can use accounting information not only to generate bills but also to tune the system and plan increases in its capacity. For the first requirement it is important that the information recorded be as accurate as possible.

There are two forms of UNIX accounting, per-process and per-login, accumulated in `/usr/adm`. The process accounting files can be massaged into per-user and per-command summaries, though this information is only available The login accounting information can be used to determine usage patterns of users and terminals. System V provides a toolkit which can be used for billing purposes, however a user can defeat it with little effort. The accounting data can quickly grow to huge volumes if it is not regularly summarised.

The System V `sar` and Berkeley `vmstat` and `iostat` utilities can be used for crude real time performance monitoring, though understanding the columns of numbers they produce is an arcane art.

User Administration

The evolution of user account management facilities for UNIX has proceeded in much the same way as for resource control, that is, by accretion of features. In spite of the provision of menu systems, a variety toolkit commands, and so on, the preferred method for account management is still `vipw`, or on System V, `vi /etc/passwd`[3].

Most user communities are arranged in some sort of organisational hierarchy, but it is difficult to model such a structure in the flat `/etc/group` file. Berkeley transformed the original UNIX group membership mechanism into a convenient structure which has more to do with file access control lists than organisational structure.

The presence of quotas, enhanced security features such as shadow password files, secure ttys, and network authority files has combined to make the administration task more complex than is warranted. The traditional approach, found in other, more 'commercial' operating systems, of associating attributes and privileges as well as accounting data directly with a user 'object' offers substantial benefits by way of simplicity.

[2]EMACS, e.g., all religious issues aside.

[3]Locking the password file is essential, but most menu systems fail to do so properly.

General requirements of a solution

The critique clearly shows that the approach to system administration to date has been characterised by an undisciplined, *ad hoc* approach, severely limited by the UNIX system's lack of a strong abstraction for a user. In the absence of an obvious place for storage of other user attributes, the growth of multiple separate control mechanisms has resulted in an almost unmanageable whole.

It is clear that a new user model is required which can be configured by the system administrator to reflect the individual requirements of systems. As the kernel itself needs to be able to access and modify user attributes (for example disk quotas), the proposed user object should be at least understood by the kernel and preferrably managed by it. Additional advantages of a kernel based implementation are higher performance and reliability by means of caching and interlocking.

A final important requirement for the user model is that it reflect the administrative arrangements of users. It should be possible to group related users (and groups) together and to administer them as a single unit. This grouping can best be achieved by arranging user objects in a tree structure, each user having a controlling user immediately above it in the hierarchy[4]. An hierarchical arrangement allows user administration to be decentralised, by means of privileges granted to controlling users.

A Unified Solution: SHARE II

Background

The original work which produced both SHARE II and Berkeley quotas was done at the Universities of Sydney and New South Wales starting in 1980. The problem of managing a UNIX system[5] with a student user population of nearly 1000 without designated operations staff was a daunting problem. Other changes to UNIX for this environment included security hardening and kernel/application performance improvements[6].

The SHARE II software is the result of ten years of this user management experience, and has evolved further to address the requirements of more traditional commercial large-system users. The software has been developed collaboratively by the University of Sydney and Softway, and is jointly owned by both parties.

Design

Per-user Information

The SHARE II system is built around a fundamental addition to the UNIX kernel: a per-user structure called an *lnode*. The *lnode* contains all the data associated with each user including resource controls, accounting data, permissions and other data.

For every unique UID defined in the */etc/passwd* file there may exist a corresponding *lnode*. Lnodes are stored on disk in a regular file and automatically transferred to and from memory on demand by the kernel. Because *lnodes* are indexed by UID, it is fast to locate and fetch an *lnode* from disk when required, though a cache is maintained in memory to eliminate disk access in most cases.

The data fields of an *lnode* are called *attributes*. All *lnodes* have the same internal structure, which is declared in a central *configuration file*, described in detail later. An attribute may have one of the following types: *short*, *long*, or *float* which correspond

[4]The logical choice for the topmost user is *root*.

[5]Originally a VAX 11/780.

[6]Many of these ideas found their way into the 4BSD distributions.

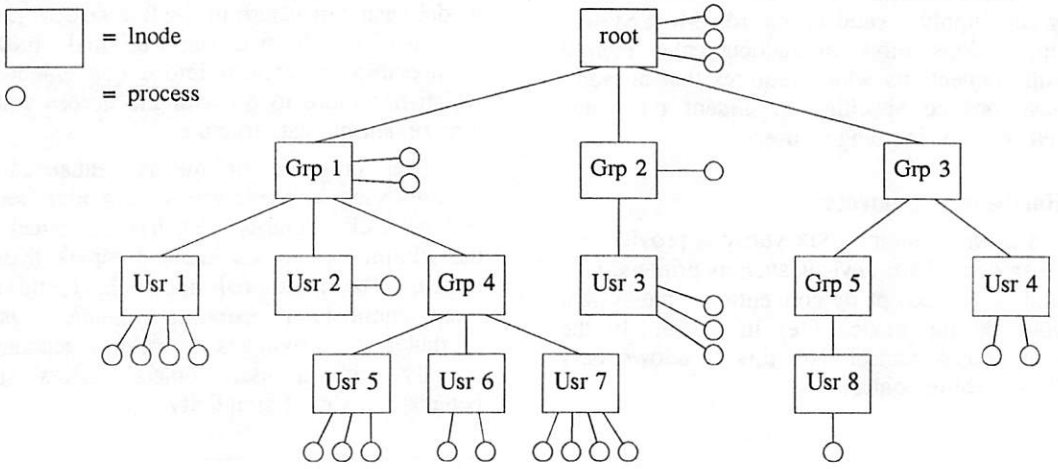


Figure 1: Example of a scheduling tree

to C data types or *huge* which is a 64 bit integer and *flag* which is similar to boolean.

Some attributes (dubbed *system*) are used directly by the kernel. These include numeric variables used to control resources such as processes, disk space, and memory size, and flags which control system privileges at the kernel level. Other attributes (*user*) are never used directly by the kernel, only by user-mode processes.

Control attributes are interpreted in such a way that the value *zero* has a default action that preserves standard UNIX semantics. A newly initialised *lnode* is completely zerofilled, so that SHARE II has no unexpected effects until administrators start altering attributes.

Hierarchical Structure

Under SHARE II, users occupy the nodes of a hierarchical tree structure called the *scheduling-tree*. This is implemented with a system attribute in each *lnode* which contains the UID of the user's parent in the tree. The UNIX kernel maintains pointers between cached *lnodes* to improve performance.

Each sub-tree of the scheduling tree is called a *scheduling group*, and the user at the root of a scheduling group is the group's *header*. The *root* user is the group header of the entire scheduling tree (Fig. 1).

The term *scheduling group* should not be confused with a *group* as defined in the `/etc/groups` file. The assembly of users into scheduling groups has no effect whatsoever on the meaning of `/etc/groups`, which remains intact.

System Calls

SHARE II is controlled through a single new multi-function system call, `limit(`*func, uid, arg*`)`. The first argument, *func*, is a code for the function being invoked. The second argument, *uid*, is used with some functions to address an *lnode* by numerical UID. The third argument is a general-purpose parameter, with different meanings for different functions.

The system call provides functions for fetching copies of *lnodes*, enabling and configuring SHARE II's kernel functions, attaching a process to an *lnode*, changing the value of an attribute in an *lnode*, and creating and deleting *lnodes*. The only way that user programs may operate on *lnodes* is via the SHARE II system call.

Semantics

Every process is attached to a single *lnode*. The *init* process is always attached to the root *lnode*. When processes are created by the *fork* system call, they are attached to the same *lnode* as their parent. A process may attach itself to any other *lnode* using the SHARE II system call, provided that it has sufficient privilege to do so. If the nominated *lnode* does not exist, an error is returned, and the calling process remains attached to the same *lnode* as before.

Configuration File

All attributes, both system and user, are defined in the SHARE II configuration file. Each definition gives the attribute's name, type and field number. In the case of flags, a flag number and a default value are also specified. The configuration file also contains definitions of device categories, device trees, and the texts of messages. The configuration file may be altered at any time, allowing the central system administrator to create extra attributes needed to control user-mode resources.

All programs which deal with attributes do so by reading the configuration file into memory, then looking up attributes by name to find their types and addresses. The kernel does not use the configuration file. Consequently, the system attribute declarations in the configuration file must conform with the kernel's built-in definitions.

The configuration file is compiled from an ASCII text source file, and contains a header, some hash tables, arrays of objects, and a string pool. The compiler optimises the hash tables so that name lookups are fast, while keeping the file as small as possible. This allows programs that use the configuration file to load and use it in minimum time.

Resource Models

SHARE II resource control is based on two different resource models: *fixed* and *renewable*. In addition, a resource can be controlled *hierarchically*.

Fixed

Fixed resources are those which are present in a fixed amount, such as disk space and memory. Fixed resources may be consumed (allocated) and relinquished (freed). SHARE II employs a *usage* and *limit* model to control a user's *amount* of usage of a fixed resource.

Renewable

Renewable resources are those which are in continuous supply such as printer pages, network packets or CPU ticks. Renewable resources may only be consumed, and, once consumed, cannot be reclaimed. SHARE II employs a *usage*, *limit* and *decay* model to control a user's *rate* of consumption of a renewable resource.

Hierarchical

If a hierarchical limit is assigned to a group header user, then it applies to the usage of that user plus the total usage of all members of the scheduling group. This provides a very flexible form of resource control, allowing limits to be placed on entire groups as well as on individual members of any group.

Resource Control Under SHARE II

This section is a brief outline of the resource controls available in SHARE II.

A fair idea of the resource controls provided by SHARE II can be gleaned from the output of the *pl*(1) utility, which prints a one-screen report of any given *lnode* (Fig. 2). Users may, at any time, invoke *pl* to view their own, or any other user's *lnode*.

The output of *pl*(1) gives information about the user's password file entry, the CPU scheduler, disc and memory resource control, privilege flags and device control. *Share* fields express the user's CPU entitlement, *accrue* fields are long-term accounting information (GB.h means gigabyte.hours), and *usage* and *limit* fields are for resource control.

Disk Space

Disk usages and limits are implemented using an abstraction called a *domain*. A domain is a set of mounted filesystems. Once mounted, file systems may be *enrolled* and *unenrolled* at leisure in any combination of domains, although this is typically done only by commands in the *rc* startup script.

One *usage* attribute per file system is set aside in each *lnode*. These attributes are kept up-to-date by the kernel whenever files increase or decrease their size, or inodes are allocated or deallocated. SHARE II provides a program for recalculating these attributes by scanning a filesystem similarly to *fsck*(8).

Several attributes for implementing fixed resource control are set aside in each *lnode* per domain. Domain usages are maintained by the kernel to be exactly equal to the sum of the usages of their currently enrolled file systems. Each domain has a *hard limit* attribute and a *soft limit* attribute. Any system call that would raise a domain usage above its soft limit will cause the user to be notified, but the system call will succeed. Any system call that would raise a domain usage above its hard limit will cause the system call to fail with an error, and the user will be notified.

```
$ pl caret
Login name:                    caret       Uid/gid: 124/50
Current logins:                    1       Sgroup:  progs (50)

Shares:       10   Share:       22.2 %     Usage:                    98209
Myshares:     10   E-share:     16.3 %     Accrued usage:        1.8901e+06

Disk usage:                  4.5811 MB     Mem usage:                1.723 MB
Soft disk limit:                 0 B       Mem limit:                    5 MB
Hard disk limit:               150 MB      Proc mem limit:               0 B
Disk accrue:        4.44105471114 GB.h     Mem accrue:       37.4133170 GB.s

Last used: Wed Jul 10 16:23:50 1991        Processes:                    8
Directory: /wrk/caret                      Process limit:               27
Name:      Neil Russell,-x107
Shell:     /bin/sh
Umask:     0002 (rwsrwsr-t)

Flags: usenet+^

Device                 Limit        Usage     Decay/Interval      Accrued
Connect-Time             16h      8h35m4s             16h/1w    1w1d17h3m
Printer-Pages            150           81              30/1d          753
$ _
```

Figure 2: Output of the *pl* utility

Note that the functionality of Berkeley disk quotas can be duplicated by having as many domains as file systems. Domains are more powerful than quotas because of hierarchical limits and the ability to group file systems together as one resource.

Memory

Memory is managed using two fixed resource models, one for virtual memory size and one for resident set size. The virtual memory limit applies to the sum of the memory sizes of all processes attached to the *lnode*. The resident set size limit is a soft limit; its effect is to bias the swapper. In addition, there is a per-process virtual memory limit that behaves like `RLIMIT_DATA` combined with `RLIMIT_STACK`. All three limits are hierarchical.

CPU

The rate of CPU service is controlled using a *fair share scheduler*. Each *lnode* is assigned a number of CPU *shares*, analogous to shares in a company. As a process consumes CPU ticks, invokes system calls, and uses I/O services, the CPU *usage* attribute of its *lnode* increases by the costs of the actions[7]. The scheduler regularly adjusts the priorities of all processes to force the relative ratios of CPU usages to converge on the relative ratios of CPU shares for all active *lnodes*. In this fashion, users can expect to receive at least their entitlement of CPU service on the long run, regardless of the behaviour of other users. The scheduler is hierarchical, because it also ensures that groups receive their group entitlement irrespective of the behaviour of the members.

The scheduler just described is a *long-term* scheduler; users who use the machine in excess of their entitlement receive poorer response than lighter users, and vice versa, until the imbalance is corrected. A method for shortening the 'memory' of the scheduler is provided by exponentially decaying all CPU usage attributes. The half life of the decay determines the character of the scheduler, and may be set by the central administrator. The scheduler incorporates safeguards to prevent marooning and ensure reasonable behaviour in extreme situations.

The primary advantage of the SHARE II scheduler over existing UNIX schedulers is that it schedules *users* not processes. This means that users cannot consume CPU at a greater rate than their entitlements allow, regardless of how many concurrent processes they run. The method for assigning entitlements as a number of shares is simple and understandable, and the effect of changing a user's shares is predictable.

[7]Configurable by the central administrator.

Processes

The number of processes that users may run at once is controlled using a fixed resource model with hierarchical limits.

Terminals

Terminal login privileges and connect-time are controlled using only user attributes to implement a renewable resource model, with hierarchical limits. The kernel is not directly involved, other than providing a simple facility for helping identify exactly when a user logs in and out.

Terminal devices are organised into groups, and flag attributes are defined to control login privileges for terminals and groups. Each terminal and group has a separate cost, and users are informed of the cost of the terminal when they log in. As a user approaches his or her connect-time limit, warning messages are sent to their terminal, and when the limit is reached, the user is notified then forcibly logged out after a short grace period.

Other Hardware Resources

SHARE II provides a complete set of library functions and shell utilities for manipulating and testing attributes. Any resource model can be implemented by declaring extra user attributes, and suitably modifying existing scripts and programs to invoke SHARE II functions and utilities. Extra attributes can be added to the configuration file by the central administrator, and there is spare space set aside in each *lnode* to accommodate this expansion.

A port of SHARE II to any system includes modifications of the local print spooler to allow control of printer pages using a renewable resource model with hierarchical limits. If other, specialised forms of resource control are needed, the central administrator may implement them directly in the same fashion as the print spooler, without requiring kernel modifications.

Accounting Under SHARE II

Every *lnode* contains an *accrue* attribute for every accountable resource[8]. The accrue attribute for renewable resources increases in parallel with the usage attribute, but is never decayed. The accrue attribute for fixed resources increases at a rate equal to the value of the usage attribute, recording the area under the usage curve over time, expressed in units of *byte-seconds* (or *megabyte-hours*, or whatever, depending on the magnitude). Accrue attributes are hierarchical, and very accurate. Note that SHARE II does not do any accounting that requires constantly growing files.

[8]That is, for every resource except number of processes.

The values in accrue attributes can be used to prepare invoices for usage at whatever points of the scheduling tree are appropriate. SHARE II does not provide billing and accounting facilities, but has a report-generating utility sufficiently flexible to allow easy extraction of accrued usages for input to any accounting system. SHARE II does provide a utility, *rates*(1), for summarising accrued CPU usages in a dynamically-updating screen display (Fig. 3).

User Administration Under SHARE II

With the addition of SHARE II, an entirely new degree of control over users can be achieved. If resource limits are assigned, then the administrator will subsequently be faced with the burdensome task of constantly responding to users' and groups' requests for changes to their allocations. Under SHARE II, the power to make such changes can be distributed to nominated users, instead of being the sole perogative of a central administrator.

There are three basic administrative privileges that can be granted to any user by setting the appropriate flag attribute in their *Inode*. Careful attention has been paid to the possibility of security holes, and it is impossible for a user to acquire any of these privileges without having had it explicitly granted from higher up.

USELIM

Users with the USELIM privilege are able to alter resource limits, create new users, delete existing users, and bestow any privilege on any user in the system, including themselves. The ADMIN privilege can be thought of as being a super user for only the SHARE II system.

ADMIN

Users with the ADMIN privilege have powers similar to the USELIM privilege, but confined to users below themselves in the scheduling tree.

SUBROOT

Users with the SUBROOT privilege have superuser access to files belonging to any user below themselves in the scheduling tree.

Conclusion

Standard-making bodies and the computing industry[9] are currently focused on network and system managability requirements. We believe that the prevailing preoccupation with the difficulties of building and standardising distributed management

[9]OSF, UI, etc.

```
$ rates -C -a
User/group  No. Proc  %Cpu                              Wed Jul 10 18:14:48 1991
System        4   31   4.2 ##=============================================      |
idle          1    1   9.1 ####                                                |
progs         3   17  54.9 ========================!==     I                   |
admin         2    4  31.8 ================I   !                               |
caret         1    8  36.0 ########!##I######                                  |
andrewb       1    3   0.1           I   !                                     |
chris         1    6  18.9 #########  I!                                       |
joe           1    1   0.0       I            !                               |
lisa          1    3  31.8 ###!#######I#####                                  |
$ _
```

Figure 3: Output of the *rates* utility

frameworks may be avoiding the issue of the suitability of existing *objects* for management of any kind.

The design of SHARE II, as elaborated in this paper, can form the basis of a significantly more powerful management system, providing a single point of access to all user attributes and increased management capabilities within an efficient and extensible framework. SHARE II adds many new resource controls to UNIX without compromising existing semantics, and allows decentralisation of administrative privileges without compromising security. SHARE II is intended to be fitted into a distributed administration environment as an agent for implementing a user object, but can also be used as a stand-alone system.

Earlier versions of SHARE II are currently available from Cray and Convex as an integral part of their operating systems. SHARE II has been successfully ported to several System V Release 4 platforms. Because SHARE II is a kernel level product, it is generally not available, except from hardware vendors to whom it has been licensed.

Author Information

Andrew Bettison graduated with a first-class honours B.Sc., majoring in Computer Science, from the University of Sydney in 1985. In 1986 he joined Fairlight Instruments as a research and development programmer. During his three years with Fairlight, Andrew worked on most aspects of the Fairlight Series III Computer Music Instrument, including user interfaces, enhancements to the operating system kernel, real-time queueing systems, numerical waveform editing and FFT algorithms, analog-to-digital conversion, and a high-performance process scheduler for the M68020. He left Fairlight and joined Softway in February 1989, and has been principally involved in the design and implementation of most aspects of SHARE II. He may be contacted at Softway Pty Ltd, PO Box 305, Strawberry Hills NSW 2012, AUSTRALIA, or by electronic mail at `andrewb@softway.sw.oz.au`.

Andrew Gollan attended the Basser Department of Computer Science at the University of Sydney from 1980, providing kernel-level support for several PDP11s running UNIX V6 and V7, participating in the port of the latter. Over the next few years Andrew was a research programmer for the University, consulted to a number of companies, and worked in a wide variety of commercial UNIX applications. He was involved in a major contract with the Overseas Telecommunications Commission of Australia as a research programmer and UNIX facilities manager. In early 1986, Andrew set up his own company, offering UNIX system configuration, prepurchase evaluation, and system programming services. Since joining Softway, Andrew has worked on several ports of UNIX System V, and is involved in the development of SHARE II. Andrew is a member of the AUUG management committee and chairman of the AUUG '91 conference committee.

Chris Maltby gained a B.Sc. from the University of New South Wales, majoring in Computer Science. During the next five years in a systems programming role at the Basser Department of Computer Science at the University of Sydney, Chris was responsible for the installation and modification of the system software for the first UNIX VAX 11/780 in Australia, and the development of the Fair Share Scheduler that was later to evolve into SHARE II. In 1986 Chris became a founder of Softway, and is now responsible for co-ordinating the Reseach and Development functions of the company. Chris has been involved with Unix since its first use at the University of New South Wales in 1975 and has worked on most versions, including Berkeley 4.2/3, AT&T System V and Edition 8, writing many device drivers and making extensive changes to the kernel. He is the Vice President of AUUG (the Australian Unix Users' Group), participates in the activities of Unix International, the Open Software Foundation, and the IEEE POSIX committees.

Neil Russell established himself in computer programming in 1982, writing commercial applications software for a firm of insurance brokers. In 1983 Neil joined TIME Office Computers as part of their communications team. Neil's duties included intelligent terminal software for several mainframes, maintenance of the CP/M operating system, and maintenance of the EON operating system (a proprietery UNIX-like operating system). In 1986, Neil joined Fairlight Instruments and was responsible for several releases of the Rhythm Sequencer software, gaining skills in the fast real-time interaction of multiple processors used to drive Fairlight's Computer Musical Instrument. Since joining Softway early in 1988, he has been involved in porting UNIX to a high-performance architecture, and the development of SHARE II's kernel modifications.

Bibliography

1. Hume, A. A Share scheduler for UNIX. AUUG Newsletter, Australian UNIX Users' Group, Sydney, Australia, 1979.
2. Kay, J., Lauder, P., Maltby, C., and Tollasep, S. The Share Charging and Scheduling System. Technical Report 174, Basser Department of Computer Science, University of Sydney, Australia, May 1982.
3. Bettison, A., Adcock, F., Chubb, P., Gollan, A., Maltby, C., and Russell, N., LIMITS - A System for Unix Resource Administration, *Proceedings of Supercomputing '89*, ACM Press, 686-692.
4. Kay, J., and Lauder, P. A fair share scheduler. *Communications of the ACM 31*, 1 (Jan 1988), 44-55.

5. Bach, M.J. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, N.J., 1986.
6. Richie, D.M., and Thompson, K. The UNIX Timesharing System. *Bell Systems Technical Journal 57,* 6 (July-Aug 1978), 1905-1929.
7. Lions, J. Experiences with the UNIX Timesharing System. *Software - Practice and Experience 9,* (1979), 701-709.
8. Computer Systems Research Group, University of California, Berkeley. *UNIX System Manager's Manual.* USENIX Association, 1986.
9. AT&T. *UNIX System V Release 4 System Administrator's Reference Manual.* AT&T, 1990.

System Resource Accounting on UNIX Systems

John Simonson - University of Rochester Computing Center

ABSTRACT

This paper describes an implementation of system resource accounting for BSD derived systems. Provisions are made for accounting for CPU time, disk block usage, connect time, and pages printed.

Our design specifications required that standard BSD utilities be used whenever possible, and that the accounting processes have a minimal impact on system performance as perceived by interactive users. We had the additional requirement that system resource data be structured appropriately for a central accounting/billing system used by the University of Rochester.

The design consists of a client-side accounting consisting of data collection and balance reporting, and a master-side accounting, where the data for each client is processed, submitted to the billing system, and where budget data from the billing system is processed and returned to the clients. The client-side requires only standard BSD utilities, while the master-side also requires *perl*.

The resulting programs have run on a variety of hardware platforms and variants of the BSD flavor of UNIX at the University of Rochester for the past two years.

Context

Most versions of the UNIX operating system provide for accounting of system resources, such as central processor time, disk storage, connect time, and number of pages of printed output. The usual utilities that provide accounting for these services do not produce output in a form that is directly useful to system administrators. In addition, the administrator must take care to ensure that accounting files used by these utilities are correctly maintained; rolled over in some cases, overwritten in other cases, and never (well, almost never) touched by human hands in yet other cases.

System V variants of UNIX and SunOS beginning with release 4.1, provide a set of scripts and binaries that assists the administrator in maintaining these files and yields reasonably useful accounting records and reports, and provides for file maintenance. The Berkeley variants generally do not provide these additional utilities. At the University of Rochester, we maintain primarily BSD type systems. While most are running SunOS (from SunOS 3.4 through 4.1.1), we also have systems running DEC Ultrix 3.0, Solbourne OS/MP 4.0C, and Alliant Concentrix 5.0. In early 1989 we developed (independent of the System V type utilities) a set of programs to use the standard BSD accounting utilities common to all of our systems, to provide a method of resource accounting consistent across all of our systems.

While several commercial packages are available to perform accounting, we felt that we would rather utilize the tools that come with the operating system. We also need to provide accounting records in a form acceptable to the university-wide accounting and billing system run on a large IBM mainframe running the MVS operating system. In most cases, we would have to write additional filters to reformat the output from the commercial package to produce MVS-acceptable input.

Our site had previously used a commercial package[1], and found it had two major flaws. First, whenever the accounting program was active, system performance took a heavy hit. Second, we found that we still had to post-process the data files produced by this package to be useful in our environment. This second point is (probably) true for all commercial system accounting packages. The net result is that there is really no gain in using any of these commercial packages, as we end up doing the same amount of additional work that we would using the default utilities provided with the OS. We do get to pay a considerable fee to use the commercial packages however.

[1]The determination that available commercial packages were unacceptable was made before the author joined the UCC UNIX OS Group. However, the author was familiar with the effects of one of the commercial packages (the executable was named *ja*) as a user on the system on which it ran (a Pyramid 90).

The University of Rochester Computing Center (UCC) discontinued system accounting on its UNIX systems for several years. In 1988 it was decided that the UNIX systems should have system accounting that conformed to the accounting performed on the MVS, VM (CMS), and VMS systems owned by UCC. It was also decided that if possible, standard UNIX (BSD) utilities should be used to collect, prepare and transmit the resource usage data. At the time we were preparing for a much more heterogeneous environment than we currently have, and wanted to avoid having to license commercial software for multiple platforms.

The first version of our accounting suite appeared in early 1989, and was on line but in test mode until May of that year. A second release in late June of 1989 cleared up some deficiencies, and changed the output format so that the output would be acceptable to a new external billing system. A third major release cleaned up some bugs, added some features, and cleaned up code, and made the code and procedures considerably more consistent among the various modules. A fourth major release is being tested, and this paper will try to indicate differences between the current release (3.2) and and the version undergoing testing.

General Overview

Initial Design and Testing

In keeping with the two primary design goals (use standard UNIX® utilities whenever possible, minimize the impact on the systems involved), we originally designed the accounting suite to run under the control of a monolithic program on the accounting master, which would fire up and retrieve data from clients via the use of *rsh(1)*. The abysmal performance (in terms of time-to-completion, system load on the local and remote system, success of process, and disaster recovery) made this approach quite unattractive. While experimenting with this method, it became apparent that using only the standard shell (Bourne or C shell) and *awk* for pattern matching, data massaging and output formatting that we had a non-optimal situation. None of our systems at the time had *nawk*, which would have helped in some of the problems we faced. However, we did have *perl* version 2.0 available and running on all of our systems. Perl was able to execute external programs and read and parse the output of those programs quite easily, and made our initial design possible. Perl also had the advantage in that it was consistent across architectures and platforms. Awk, while available on all of our hosts, was not sufficiently consistent in it behavior. For example, the awk on our super-mini computer frequently core dumped. It also had truncation problems with decimal values with more than four digits significant to the right of the decimal place. The same data processed on this host and others often produced very different results when processed with awk.

So perl became an important part of our suite. Since perl was rather new at that time, we decided to hedge our bets. We decided that the collection of raw facility usage data and selection of that data could be performed using standard UNIX utilities. The final processing of the data would be handled by perl on a central machine. We decided that system resource accounting at our sites would have minimal performance impact and maximal reliability if we did the following:

- Make each system which was to have resource accounting responsible for only three tasks, collecting the data, maintaining the data files, and transmitting the collected data to a designated host for further processing.
- The data collected at the individual sites is processed on two or more central hosts. All of these hosts are responsible for managing the data files delivered by the individual systems, processing them, and cleaning up afterwards. One host is designated the primary host, and should send the processed and formatted output to a remote billing system. The remaining central hosts are considered secondary, and should not send processed data to the remote billing system. They should however run a rescurce usage reporter program and save that report.
- Run both the client (data collection) and master (processing and formatting) processes at off-peak hours.
- Run the master processes on "non-interactive" hosts.

Our testing indicated that the first two stipulations are in fact important. Experience has shown that this approach results in excellent reliability (and disaster recovery), and system performance is not noticeably affected. Failure of a single client only affects that client, and that failure is usually recovered when the system comes back up and accounting is run again on schedule. Failure of an accounting master can be recovered by transmitting data from a secondary master to the billing system.

The final two stipulations are probably not as important. The impact on the system is such that we could run the system accounting programs anytime during the day without fear of complaints about poor system response. The time to run varies of course due to inherent cpu speed, cpu load, amount of disk involved, number of records to be processed or collected, and so on. Most systems complete the data collection in one minute or less (elapsed time, not cpu time), while the large file server usually takes three minutes or less to collect its data (which includes running *quot* on over five gigabytes of disk. The master-side usually takes less than four minutes to complete (on a VAXStation 3100 or a Sun 4/260) processing about 3500-5500 output records. We do

run the master-side processes on "non-interactive" hosts (one is a file server, with staff-only logins, the other is a nameserver and DECNET link for printing, again with staff-only logins), but do this more to keep the data and processes safe from prying eyes than to minimize impact on the system[2]. These service oriented machines also tend to be more reliable and stable than most of our cpu servers.

Current Design

In our current implementation, we have chosen to split accounting duties into four major components. The first of these is the collection and initial preparation of the raw resource usage statistics. This activity is carried out using the standard BSD utilities of *sa(8)*, *quot(8)*, *last(1)*, and *pac(8)* on systems referred to as *accounting clients*. *Rdist(1)* (or *rcp(1)*) is used to transport copies of the collected data to the accounting masters.

While our choice of *sa* for CPU accounting, *quot* for disk accounting, and *pac* for printer accounting is rarely questioned, we have been asked about our choice of *last* over *ac(8)* for connect time accounting. The reason is that we lose too much information when we use *ac*. We would need to roll over **/usr/adm/wtmp** nightly, and would lose our history of who was on the system and when they were on, and we would also lose the information about which tty line was used, and if a pty, which host the user came from. This calls into question our choice of *sa*, as that also requires that the data accumulation file (**/usr/adm/acct** or **/usr/adm/pacct**) be truncated nightly. We lose the ability to track command usage (via *lastcomm(1)*) for any activity prior to the current day.

The second major component is carried out on systems referred to as *accounting hosts* or *accounting masters*. These systems process the raw usage statistics produced on the accounting clients producing *mvs records*, records that are ready for input to the master billing and accounting system on the MVS system. The accounting host is also responsible for transmitting these records to MVS.

The third component is the MVS processing. At this time, the **KOMAND** accounting and billing package is used on MVS. Data from UCC hosts flows to this system, and billing reports (and balance sheets) are returned to the UCC hosts.

The final stage of processing is the *refeed* processing. These are the records returned to Unix from the MVS billing system. These records are processed on a single host (usually, but not necessarily, the primary accounting host) referred to as the *refeed host*. On the refeed host, the data are appropriately redistributed to each accounting client. Each accounting client processes that data and generates a *dbm(3)* database for fast access by the *balance(1ucc)* program.

At sites not reporting to the central billing system, the third component can be replaced with the use of the charge calculation and reporting program that is part of the accounting suite. We have not developed a program to take the output from this program (or the output produced by the second component) and feed it to the refeed processor. We have not had a need for this, although it could (and should, and will) be developed.

The charge calculation and report program is quite useful, and can produce a variety of reports for sites not wishing to deal with billing. See the **Examples** section for more information about the charges reporter.

Performance

We have been running this system resource accounting suite on a variety of systems since early 1989. The current host load consists of a Sun 3/280S (800–1700 password file entries), five Sun 3/60 diskless clients served by that 3/280S, a Solbourne 5/602 (540–1200 password file entries), an Alliant FX/80 (under 500 password file entries), a Sun 4/260 (staff-only interactive use, but providing home directory file service for the hosts listed above, and file service for licensed and home-grown software for approximately 60–80 hosts), and a VAXStation 3100 (staff only use, but was the primary nameserver, printserver, and accounting master for the cc.rochester.edu domain). The accounting suite was also run on two VAX 750's running More/BSD and a VAXstation 2000 running ultrix 3.0 until those hosts were retired.

In addition, a second cluster of hosts maintained by our group (an additional 8–10 hosts, Sun 3/80, 3/60, 3/160, 4/260, and 4/65 platforms) has been running this suite. This group of hosts does not send accounting data to the external billing host, and relies on the *getcharges* program included with the accounting suite.

Disk Requirements

The programs and configuration files for the accounting suite (release 3.2) require about 300 Kb (not including the userid and account number registries). Client-side materials are about 55 Kb, master-side materials about 145 Kb, and materials common to both are about 37 Kb. The refeed processing materials are about 50 Kb. Release 4.0 of the accounting suite should not exceed 400Kb.

Data storage requirements will depend on the level of system activity and the number of users, and on how often you are willing to archive data to tape.

[2]The University requested that a user's budget and expenditures remain moderately private. Only the user and privileged users may view the accounting information for that user.

Given the configuration of hosts described above, we can figure on disk usage for the accounting master as follows per week (compressed files marked by *):

File Type	Files per Week	Avg Size (Kb)	Total Kb/Week
*outbound data	7	45	315
*sentrecs	10	150	1500
*culls	10	3	31
charge report	7	240	1700
refeed data	7	500	500
Total			4046

Figure 1: Estimated disk space requirements

While the 500 Kb refeed file appears each day, it is overwritten each day, and thus the average size and total Kb needed per week are the same. Currently, our two accounting masters each have approximately 5.5Mb of data, while our accounting clients have between 150 and 350Kb of data each.

The client-side disk requirements are of course considerably less. Since the file maintenance methods for the clients results in an automatic rotation, system guano[3] is kept to a minimum, and disk needs do not get out of hand. Weekly requirements (rotation is based on a weekly scheme) range from about 32 Kb for a diskless client that is not heavily used, to about 300–1000 Kb for a heavily used system with local disks.

CPU Requirements

As stated elsewhere in this document, the programs in the accounting suite have very little impact on perceived system performance. The client-side accounting processes usually complete in under one minute elapsed time, with three to four minutes considered a long time to complete the tasks. Disk accounting data collection (using *quot*) usually consumes the largest part of the elapsed time in client-side accounting. Collection of connect time data records is sensitive to the size of the **/usr/adm/wtmp** file, and may also take a large portion of time. Printer accounting and cpu accounting usually run very quickly.

Master-side accounting takes a bit longer, but usually completes in under four minutes, and rarely takes over five minutes. The accounting processes seem to be affected more by system load than they contribute to that load. Again disk accounting seems to take a large portion of the elapsed time, because of the large number of records involved. Connect time accounting is the most labor intensive portion of the processing (on a per-input-record basis), due to the amount of parsing, time calculations, and record selection that needs to be done. Raw data record parsing is fairly simple for disk, cpu, and printer accounting.

[3] *System guano* is our local term for the debris that collects on a system – files whose purpose and usefulness are unknown. Good procedures (and use of rcs or sccs) can minimize this type of cruft.

Disaster Recovery

Disaster recovery has proven itself to be easy and straight forward. A client that crashes and fails to send data will catch up the following night. Disk blocks per day cannot be recovered, but that is not a problem, since the users could not access those disk blocks during the downtime. A failure of data transmission (versus failure of data collection) only becomes critical when the autorotation will destroy week-old data files that have not yet been transmitted. Transmission failure is usually due to the accounting master host being down, and the files can be transmitted to that host when it comes up, and the accounting suite can be invoked manually on the master.

We have outlined in our operation notes, included with the software, what to do for a variety of failures. A low-priority item on our to-do list is to transform the written instructions to a script that will automate the task of disaster recovery.

Data and Command Flow

This section is a brief walk through of the accounting process, for both accounting clients and masters.

Connect, CPU, and Disk Accounting (Client-side)

The commands involved in this process are *SysAcct(8ucc)* *CpuAcct(8ucc)*, *DiskAcct(8ucc)*, and *ConAcct(8ucc)*. The process is:

1. Set up for the actual processing. Change to the correct directory, set a timestamp, get the node name for this system, and create a list of files to *rdist(1)* to the accounting master.
2. Prepare a list of file systems. Either parse *df(8)* output with *awk(1)*, or read the file **fs.include** if it exists. The file **fs.exclude** (if it exists) indicates which file systems (if any) should be explicitly excluded from disk accounting.
3. Prepare a set of empty data files, with correct owner, group, and permissions.
4. If the file **PreAcct** exists, assume it is an executable to be run *prior* to data collection for accounting on this system. This file is *not* part of the distribution, and is something that a particular system administrator will create when needed. An example of this type of executable is a script that uses *chown(8)* to change ownership of all the formatted man pages to some standard system userid (e.g., bin or root), so users are not charged for having looked at a man page. The actual

PreAcct file should reside in some private area, and the "file" examined/executed should really be a link pointing to this file. At our sites for example, **/usr/ucc/etc/PreAcct** is a soft link pointing to **/var/ucc/etc/PreAcct**.

5. Begin cpu accounting data collection, by writing a timestamp record into the cpu data file (**cpu**). Locate the *sa(8)* program, and execute it so that the information in any summary file is ignored, so that there is no interactive threshhold compression, and to display the number of processes and number of cpu minutes per user. Append the output to the cpu data file. Finish by immediately running the *sa* command again, this time collapsing data into the summary files **/usr/adm/savacct** and **/usr/adm/usracct**. This output is discarded. It is imperative that you ignore the data in the summary files, and that you then collapse the current **/usr/adm/acct** file into the summary files. Failure to meet these two requirements means you would always include all cpu usage for a user since that person started using the system (actually, it would go back to the oldest time recorded for that person in the acct or summary files, but it would continue to charge a user repeatedly, rather like the electric company never zeroing out your electric bill -- each month's bill would be the sum of this month and all previous months).
6. Run disk accounting data collection. Write a timestamp record to the data file (**dsk**), and then find the *quot(8)* program. Run *quot* on each file system in the list created in step [2], above. Append the data to the disk data file, ensuring that only data records are written. system in the list created in step [2], above. Write the data to the disk raw data file.
7. Run connect time accounting, by running the *ConAcct(8ucc)* script. Standard output and standard error get cleaned up so that bogus error messages are removed (e.g., our Alliant FX/80 likes to complain on occasion about "too many terminals"). ConAcct first checks to see if a **LastRun** file exists. If not, it creates one (it is a timestamp record), prints a warning message, and quits. If **LastRun** exists, create a timestamp record file named **ThisRun**, by placing a copy of the last line of **LastRun** in **ThisRun** as the first record. This defines the beginning of the temporal window for connect time accounting. Then create a timestamp record for the current time, and insert that as the second record in **ThisRun**, to mark the end of the temporal window. Dump the contents of the new **ThisRun** file and the output from *last(1)* into an in-line *awk(1)* script that performs some record selection and cleanup. The output from this awk script is directed into the connect time data file (**conf**). The **ThisRun** file is copied over the **LastRun** file.
8. If the file **PostAcct** exists, it is assumed to be an executable file (binary or script) that will provide special local processing of the data files just created (e.g., remove connect time data records for rlogin/telnet sessions from the local cluster to this host). Print a message indicating that we are about to call the local procedure, and another when we return. Like **PreAcct**, this file should really reside in a private area, and the file examined by *SysAcct* should be a link pointing to this private copy.
9. Send the files **con**, **cpu**, and **dsk** to the accounting masters (these are defined in the distfiles used), specifying the destination directory as the macro **DEST** on the command line. For systems such as our Alliant FX80 (running Concentrix 5.0), you will have to use *rcp(1)* rather than *rdist(1)*.
10. Rename the three data files (**con, cpu, dsk**) to have a suffix of *.day*, which is the three character representation of the day-of-the-week (Sun, Mon, ..., Sat). This ensures a self-rotating set of files (on a weekly basis). While a given data file is only available until accounting runs on that same day the following week, the transmitted files are available on the accounting master and on archive tapes.

Printer Accounting (Client-side)

Printer accounting is not run on all accounting clients, and is controlled by a separate script, *PrtAcct(8ucc)*.

1. Set up for processing; change to the correct directory, verify who we are, build a list of printers for each printer type for this run on this host. Create an empty data file for each printer type.
2. For each printer type, process the printers of that type using *pac(8)*. Then run *pac* again with the *-s* option to zero the data accumulation file for that printer. Rotate the summary files for that printer.
3. Send the data files (one for each printer type) to the accounting master(s), and attach a date suffix to each of the data files.

Record Selection and Generation (Accounting Master-side)

This is a simple walk-through of the accounting process on the accounting master.

1. *OverLord(8ucc)* is run directing its output to **~account/OverLord.log**. *OverLord* runs the perl scripts for connect, cpu, disk and printer accounting for each host defined in the

account.conf(5ucc) file.

2. In general, the perl scripts behave as follows. Initialize a lot of variables setting up the date, data and output file names, read in the **account.conf(5ucc)** file, read in the function libraries *accounting.pl(3ucc)* and *timetools.pl(3ucc)*. Get long and short hostnames of the client, and setup arrays connecting userid names, userid numbers, and account numbers by reading in the userid and account number registry files. Then read the raw data for the facility (con, cpu, dsk, prt), process and toss good records into the **~account**/*node*/*facility*.**mvs.***mmdd* file, and the bad records into the file **~account**/*node*/*facility*.**badrecs.***mmdd*. Warnings about good and/or bad record counts outside of limits specified in the **account.conf** file will be printed. Some events on a system register in accounting data, but are too small to deal with; e.g., cpu accounting may send over a cpu usage record that indicates an amount of cpu time used that is too small to generate a charge. This record would be discarded. Early versions of accounting scripts threw these records into the bad record file; the current correct behavior should be to simply discard these records. In fact, this "granularity" loss is now minimized, in that all facility usage records from the raw data file are accumulated into as few output records as possible. Bad records are those for which we cannot find a valid, active account number for that client for that userid name. Disk accounting can generate data records with uid number identifiers rather than userid names; if the uid number on these records cannot be matched to a userid name, the record generated will be placed in the bad records file. The next release will allow for a third class of output data records, those that are semi-valid (e.g., closed accounts/logins whose disk files have not been removed from the system), which should not be placed in the 'bad records' file.
3. On completion of the perl scripts, the raw data files are renamed to have a suffix of *.mmdd*, and control is transferred to *Morning(8ucc)*.
4. *Morning* sets up by creating a date variable in the form *mmdd* (described elsewhere in this document), a mail address variable, and primary accounting host name variable. The **account.conf** file is referenced to create lists of hostnames. The list of hostnames is sorted, and duplicate entries removed. A zero-length file to hold a tally of number of records of each type for each accounting client is created. For each hostname in our list of names, we perform a cleanup and summary process; use *wc(1)* to count the records for each data file type for the accounting client (raw data, good and bad mvs record files) and save that in a file named **count** in that client's data directory. Filter the *wc* output to remove the date stamp suffix from the file names it displays. Then append this output to the summary file. If the file **culls.tar** exists, use *tar(1)* to append the file of bad mvs records to the tar file. Otherwise, create a new **culls.tar** file. Then remove the bad mvs records file(s). Repeat this process for the tar file **sentrecs.tar**, storing the raw data records and the good mvs records rather than the bad records. Use *cat(1)* to append the good mvs records to **../outbound.***mmdd*, and then remove the raw data and mvs-ready data files. If the current host is the primary accounting master, the **outbound.***mmdd* is transmitted to the external billing host. The method of transmission depends on the nature of the beast at the other end of the connection. We need to get a copy of the data to an MVS system, so we mail to the address set in the mail address variable with a agreed-upon subject line. The verbose flag is supplied to mail, so the logfile will contain an indication of whether or not the data were successfully sent. Then the summary record count file is mailed to the accounting administrator (not the *account* userid). Finally, the **outbound.***mmdd* file is compressed. If this is not the primary accounting master, the **outbound.***mmdd* file is compressed, and any **outbound.***mmdd* files older than 14 days are discarded. Just prior to compression, the data is piped through sort (sorting on userid name) and fed to the *getcharges(8ucc)* program, to produce a file **~account/charges.***day*, where *day* is the three-character representation of day-of-week (Sun, Mon, ..., Sat). In either case, the summary file for record counts is removed, and a message saying "Accounting has completed." is printed.

In release 4.0 of the accounting suite, the tracking of number of records will be changed. Although a command line option will allow this feature to be disabled, output from the master-side perl scripts will be structured to provide the same information is a format that will be easily processed by *watcher(1ucc)*[4], which will generate the anomalous condition alerts as necessary. Good and bad records can be monitored both for upper and lower limits and for percent change by *watcher*.

[4]See Ingham, 1987, for more information about watcher.

Refeed Processing Data and Command Flow

The external billing system (locally KOMAND III V322 running under MVS) mails the processed data to the userid 'account' at the refeed host. Account's **.forward** file passes all its mail to the program *generator(8ucc)*, which examines the mail header to see if it is reasonable to accept and process this data. The "From:" line is checked for a valid originating address, a simple test to see if multiple "From:" lines are present is performed (only one "From:" line is allowed), and the date of the mailing and the date of the last file processed are checked; the current input must have a date newer than the last file processed, to keep the balances in sync. *Generator* creates a file called **~account/tmp/combined.file** containing the results of using *join*(1) on selected fields from the mailing and the *UCCacctnum*(5ucc) (the account number registry) file. This file is then fed via *rsh*(1) to each accounting client, and that rsh runs a *generate*(8ucc) command to process the stdin and produce an *dbm*(3) file *balance*(5ucc) (actually, two files, **balance.dir** and **balance.pag** are created). Only records pertaining to that client in the input are used to create the database on that client. The file *system.configuration(5ucc)* is read to determine which hosts shall have data prepared and piped to the rsh.

The current form of refeed processing has some distinct advantages. First, it is a purely interrupt driven model; the balances are updated as soon as it arrives from MVS. Data arrives at one location only, making management a bit easier. The *dbm*(3ucc) file provide very fast data access by the *balance*(1ucc) program.

The current method does have disadvantages however. It is all too easy to spoof as a user/host using sendmail, and this leaves a large security hole in our accounting. The dbm files, while fast, require that each client process the data, which is somewhat redundant and wastes cpu time. The use of *rsh*(1) is not entirely desirable. These holes will have to be closed in future releases of the refeed processing segment of accounting, as new flags in the MVS refeed data will signal that accounts should be closed, terminated, expired, reopened, and so on.

We are currently re-writing the refeed processor in *perl* and should see both an increase in speed and capability.

Examples of Configuration and Report Files

Configuration Files

Figure 2 shows a portion of section 1 of the accounting configuration file, under release 4.0 of accounting.

Release 4.0 of accounting consolidates several configuration files into a single configuration file. Figure 3 shows the second section of the configuration file, which deals with printer accounting.

The third section of the configuration file contains miscellaneous information. Figure 4 presents an example of this third section.

The use of tags (e.g., '@') may be replaced by a more verbose but easier to understand flag (e.g., SERVER, MASTER, CLIENTS, and so on). The release 4.0 version of the *getcharges*(8ucc) relies on this file for rates and units as well as host and service information. Note also the rate field often consists of two comma-separated entries. We have made provisions for different charging rates (non-university clients get charged the second rate) and this could be extended to permit peak/non-peak charging schemes.

Figure 5 shows an example of the alias file used to simplify our account number registry (explained below). If a user has no explicit entry for the host Enterprise in the account number file, that user's account number on the host Uhura will be used.

```
#host:service:type:lognumber:mingood:maxgood:minbad:maxbad:rates:units
UHURA:CON:429:220000:50:650:0:1:0.0134,0.0268:Connect Minutes
UHURA:CPU:428:210000:25:500:0:1:0.0667,0.1334:CPU Seconds
UHURA:DSK:430:230000:400:1200:0:1:0.015,0.030:Hektoblocks/Day
ENTERPRISE:DSK:450:430000:600:2000:0:1:0.010,0.20:Hektoblocks/Day
RAND:LPPRT:400:9940000:0:500:0:1:0.02:pages
RAND:LWPRT:399:9950000:0:1000:0:1:0.06:pages
```

Figure 2: Section 1 of accounting configuration file

```
# +:host:account numbers ('exception accounts')
# =:host:printer type list (list must agree with section 1)
+:RAND:ARUT65
=:RAND:LPPRT,LWPRT,LAPRT
```

Figure 3: Section 2 of accounting configuration file

Figure 6 shows an excerpt from the account number registry file. The format defines 6 colon delimited fields, account number, userid name, hostname, expiration date, lockable account (y/n), and account status (active,dead,inactive). The registry is very important both for our accounting and adduser programs.

An excerpt from our userid name/number registry appears in Figure 7. The entries in this file consist of 3 colon-separated fields, a five-digit userid number, the corresponding userid name, and the full name of the user as it appeared on their account

```
# a bang (!) indicates a list of accounting
# masters for this host follows
# !:thishostname: master1 master2 ....
!:SAREK: rand.cc.rochester.edu enterprise.cc.rochester.edu

# a percent (%) indicates host,printer type,
# and print queues of that type
#%:host:type=printer1,printer2
%:RAND:lnprt=hobbesps,hobbesa
%:RAND:lwprt=towne,taylor,classlw

# an at-sign (@) indicates a system name and which
# facilities should be done or skipped.
# @:clientname:facility requests
@:RAND: +print
@:ENTERPRISE:-cpu -con
@:UHURA:
@:SAREK:-disk
```

Figure 4: Section 3 of accounting configuration file

```
#real name:aliased to name
sarek.cc.rochester.edu:uhura.cc.rochester.edu
enterprise.cc.rochester.edu:uhura.cc.rochester.edu
```

Figure 5: Accounting aliases file example

```
acct07:aa2cs252:enterprise.cc.rochester.edu:910830:Y:A
acct07:aa2cs252:sarek.cc.rochester.edu:910830:Y:D
acct07:aa2cs252:uhura.cc.rochester.edu:910830:Y:A
apdt05:jonl:spock.cc.rochester.edu:910630:Y:D
apdt05:mpsa:spock.cc.rochester.edu:910630:Y:D
megt24:allan:enterprise.cc.rochester.edu:910630:Y:A
megt24:allan:sarek.cc.rochester.edu:910630:Y:A
megt24:allan:uhura.cc.rochester.edu:910630:Y:A
megt24:chok:enterprise.cc.rochester.edu:910630:Y:A
megt24:chok:sarek.cc.rochester.edu:910630:Y:A
megt24:chok:uhura.cc.rochester.edu:910630:Y:A
```

Figure 6: Account number registry excerpt

```
00099::
00199:jonl:Jong-Rim Lee
00200::
00327:mpsa:Malcolm Savedoff
00810:allan:Allan Schindler
01814:chok:Dawn Chock
02575:aa2cs252:cs252 Student
04014:uccdummy:Reserved for future UCC use
```

Figure 7: Uid# to login name mapping file excerpt

application. While the user may change their GCOS information in the password file, they cannot change this entry. The full registry includes records for each valid userid, from lowest to highest. Unused userid numbers have null name fields. Some numbers are locked out for future use by the systems staff. Our adduser program references and updates this file.

Accounting Report Writer Examples

The following is an excerpt of the default type of report generated by the *getcharges*(8ucc) program.

The *getcharges*(1ucc) included in release 4.0 of accounting has much more extensive report generation capabilities, as the 'usage' message displayed in

```
Userid   Account Service or Resource      Units Consumed              Charges
-------------------------------------------------------------------------------
         ARUT65  Laser Printing          596.00  Pages               $  35.7600
         ARUT65  Total Unix Charges                                  $  35.7600
ALLAN    MEGT24  Sun4(Enterprise) Disk    62.25  Hectoblocks/Day     $   0.3100
ALLAN    MEGT24  Sun4(Troi) CPU           38.40  Seconds             $   1.5300
ALLAN    MEGT24  Sun4(Troi) Connect       17.00  Minutes             $   0.2800
ALLAN    MEGT24  Total Unix Charges                                  $   2.1200
AAD2_LTD BBPT10  Sun4(Enterprise) Disk     0.05  Hectoblocks/Day     $   0.0000
AAD2_LTD BBPT10  Total Unix Charges                                  $   0.0000
             . . .
    (much deleted)
             . . .
                 UNIX Accounting Summary

Total charges for Sun3(Uhura) CPU          :   678.3700
Total charges for Sun4(Enterprise) Disk    :    75.3900
Total charges for Sun4(Enterprise) Connect :    31.4300
Total charges for Sun3(Uhura) Connect      :   138.2000
Total charges for Sun4(Troi) CPU           :   344.4900
Total charges for Sun4(Troi) Disk          :    22.8600
Total charges for Sun4(Enterprise) CPU     :   137.9200
Total charges for Laser Printing           :    55.2000
Total charges for Sun3(Uhura) Disk         :    39.5100
Total charges for Sun4(Troi) Connect       :   122.2000

Total for all UNIX computing services:    1645.5700
```

Figure 8: Sample output from getcharges(1ucc)

```
Usage:
getcharges [-a] [-d] [-h] [-t] [-u] [-A] [-H] [-D "date"] [-c conf] [-f data]
        -a:          summary by account number
        -d:          detailed summary of user/code/account/host (default)
        -h:          summary by host
        -t:          summary by service code number
        -u:          summary by userid name
        -A:          summary by account number and service code number
        -H:          summary by host and service code number
        -D "date":   specify an alternate Date/Time to put in header
                     (date must be in quotes)
        -c conf:     specify an alternate configuration file, 'conf'
        -f data:     specify the mvs-format data file, 'data'
                     if no data file is specified, a display of the
                     configuration file is produced.
```

Figure 9 - getcharges(1ucc) usage message

Figure 9 shows.

Conclusions

Why do we run accounting (other than "because the administration says you will run system accounting")? We have very few paying customers on our hosts. The university designates a fixed amount of money for the support of the computing center, and Dean's Funds (aka funny money) are allocated to departments for use in obtaining computing center services. These Dean's Funds as spent on computing center services constitute "votes" for services provided; we can show how we (the UCC) earn our keep. Setting rates for services and reporting facility usage in terms of dollars has two benefits. First, it allows a common base for calculations, i.e., we can say that ten megabytes of disk storage per day costs as much to provide as does (e.g.) forty seconds of cpu time on a Sun 3/280. Second, facility consumption expressed in dollars is much easier to understand than cpu/seconds or hektoblocks/day. Our rates are set so that we recover our costs, and may be adjusted during the year. While most of our users pay for services via Dean's Funds, we are prepared to accept real money for those services.

We can also control system usage more easily. Those users who need to accomplish work but would rather play tiny-MUD have to decide which is worth their limited amount of funds. Since our mission is to provide resources for research, university-sponsored classes, and general computing (in that order), we prefer the former over the latter, but allow the users to choose how to spend their allocation.

A side benefit of running accounting on our systems has been increased security. Since every cpu/second, connect time minute (including ftp connections) and every disk block is accounted for, unauthorized usage becomes more evident and easier to track. Clearly this is not as powerful as system auditing, but is less labor intensive (both machine and human) while still providing some benefits.

While we have tailored this system to work in our environment, it could be configured easily for any BSD environment, and probably many System V environments. Most sites would not need the refeed processing, as they do not interface with an external accounting and billing system. For those sites that do not need this facility, the built-in charge calculation program should suffice, and could be adapted to behave as if it were the provider of refeed information. Our backup accounting master host in fact runs the charge calculation program rather than sending the data to the university-wide billing system. We use our locally generated reports to periodically verify the reports generated by the remote billing system.

The built-in charge calculation and report program (*getcharges* (8ucc)) is fairly capable in accounting release 4.0. It can report by userid, by account number, by host, by facility, or by combinations of these headings.

Sites that need to interface with an external accounting/billing system different from the one we use (KOMAND III, V322, by Pace Applied Technology, Inc.) would need to rework the refeed processing, and would probably need to revise the output format for data sent to that external system.

In general, we've been pleased with the performance and reliability of this suite of accounting programs. It does its job using standard BSD utilities (with the exception of perl), and has worked in a variety of BSD environments. The update from version 3.2 to version 4.0 is designed to make the system more flexible and configurable. Its impact on system performance is minimal. While we normally run it at night, we have run the processes during the day in 'disaster recovery' situations, and its impact on performance was not (subjectively) noticable. The work load is distributed between master and client hosts, and reliablity has been quite good. Disaster recovery has been quite easy.

Changes, or What We Would Do Differently

While we are pleased with the design and performance of this accounting suite it is not perfect Expanding it to a wider installation base will quickly point out any flaws not evident in a parochial installation. But it has (so far) worked well, has been easy to use, recovered well from problems, and distributed the workload fairly well.

Most deficiencies in earlier releases have been corrected. The next major release (after 4.0) will have more of the code written in *perl*. The refeed processor is a good candiate for this, as are the primary control scripts run on the accounting masters.

We really need to have a more effective, secure, and configurable transport mechanism to/from our external billing system. Simple email has been sufficient for our needs. The MVS side expects only one mailing per night from us, and warns us if anything unusual happens. We know how many records we sent the day before, and how many they received. We do feel that if we use email for moving data we should use some form of authenticated email. Our MVS system recently added TCP/IP services, so we may be able to make use of some method other than email for data transfer. We would at least like to make the data movement method configurable (email, authenticated email, rdist, ftp, etc.).

The code, especially for the *perl* (1ucc) scripts, could be cleaned up, and streamlined. They work, and some parts are really nice, but other sections look awful. Also, they could become more

consistent in style.

The code and Makefile and installation process should be generalized to allow non-ucc sites to use the code easily. Larry Wall's *Metaconfig* could be helpful here.

The latest implementation may be a bit memory intensive on systems with large numbers of users. The associative arrays in the *perl* scripts could easily be replaced by creating temporary *dbm* files. This is more disk intensive (space and access) but may be useful in small memory machines.

We need to be able to deal with sites using Sun's NIS (formerly YP); we have not run these programs in an NIS environment. We would like to be able to deal with clusters of hosts which may not have consistent uid number to uid name mappings. We also need to improve both efficiency and error detection in the refeed processing, and to produce the final component (ledger/budget database maintenance) to make this a completely stand-alone resource accounting and billing system.

Availability

The sources to release 4.0 of this accounting suite will be made available for anonymous ftp from **cc.rochester.edu** (128.151.224.6), in **pub/ucc-src/acct-4.0** when completed. The code has been written and tested as of early July 1991. We need to complete the changes to the documentation, and update the refeed processor to incorporate new flags sent from the KOMAND package on MVS. We expect to complete the work by mid-September to mid-October, 1991.

Acknowledgements

While I have to take the blame for this paper, the overall design, and much of the code, several members of our workgroup, past and present, should be acknowledged for their contributions. Craig McGowan suggested the project early in 1987. Denise Ondishko said "Make it so, Number One", in late 1988. Scott Leadley wrote the initial connect time processing scheme, and early versions of the perl support files to make it work. Kirk Anne wrote the refeed processor and the balance program, and Mark Sirota corrected some problems with dbm file data alignment that appeared once we started using the balance program on sparc-based platforms. Millie Morton has taken on the odious task of maintaining the current version.

Thanks also to Kenneth Ingham for making *watcher* available, and especially to Larry Wall for *perl*, the existence of which has made this project viable.

Author Information

John Simonson is a senior systems analyst for the University of Rochester Computing Center. He received a B.A. from the University of Minnesota in 1974, and a Ph.D. in Developmental Psychology from the University of Rochester in 1985. Reach him via U.S. Mail at University of Rochester Computing Center, 727 Elmwood Avenue, Rochester, NY 14620. Electronic mail may be sent to gort@cc.rochester.edu, or rochester!ur-cc!gort.

Bibliography

[1] Ingham, Kenneth, "Keeping Watch Over the Flocks by Night (and Day)", Proceedings of the Summer 1987 USENIX Conference, pp 105-110, 8-12 June 1987.

[2] Wall, Larry and Schwartz, Randall L., *Programming perl*, O'Reilly and Associates, Inc., 1990.

A Next Step in Backup and Restore Technology

Rob Kolstad - SunSoft, Inc.

ABSTRACT

Every site with significant secondary storage seems to invent a new way to sequence backups. This article discusses an evolutionary step toward bringing backup and restore into the era of large disk systems.

The Problem

Every UNIX site must protect its data through some kind of backup mechanism. The data must be protected from three kinds of loss: media loss (i.e., disk failure), user error (e.g., accidental removal), and catastrophe (e.g., a tornado destroys the entire building). Note that solutions such as disk arrays mitigate only one kind of data loss. Thus, backup is a way of life at UNIX sites.

Unfortunately, UNIX backup programs seem to suffer from a common set of problems. These include:

- Inability to work correctly with partitions mounted and in use (i.e., offline dumps are intrusive to the user community)
- Requirements for extensive operator intervention
- Lack of a good catalog of which files are on which tapes
- Poor end-of-media handling
- Slow *dump* speed
- Slow *restore* speed
- Tape management
- Inability to sequence automatically through several local and remote partitions

Third party vendors have addressed some of these problems but none has been able to solve all of them.

This paper, along with several subsequent papers in these proceedings, describe the SunSoft Backup Copilot backup system that attacks and solves these problems.

Requirements for the Dump Program

The *dump*(1) program is the heart of a backup system. Whether one uses exactly *dump* (sometimes known as *ufsdump*), *tar*, *cpio*, *bar*, or some other program, the goals are similar: copy a set of files (or an entire partition) to a backup medium in hopes that the files can later be read back if needed.

One major problem with dump programs, however, is that without kernel support (or a new file system), ensuring the validity of a dump tape is a difficult matter indeed. In fact, analyzing the behavior of a dump scheme when presented with moving subtrees inside the file system structure often exposes problems of safety. The popular *dump* program, when run with a file system mounted, can get into trouble when inodes contain a directory when the first scan occurs but are re-used in the interim and become regular files. It can produce a tape which is unreadable by *restore*.

Of course, all online schemes must contend with files being written precisely while they are being dumped. TransArc's Andrew File System product solves this problem by creating copy-on-write inodes - but this requires that an entire new file system be used on your disks. Otherwise, backup programs do they best they can, sometimes warning the operator when a file was modified during backup, sometimes silently failing.

The above remarks are not as a tirade against breaking the rules and running traditional *dump* program while file systems are mounted. The probability of some of these kinds of failures is very small - nonzero, but small. The risk is often acceptable when weighed against the cost of file systems being unavailable (often for hours). I have administered several sites which have used *dump* in online mode and never had a problem. I have also heard stories of sites that did get occasionally bitten by taking the chance.

At any rate, it is desirable to make certain promises about the quality of backup tapes produced. Programs with stronger guarantees (e.g., "never creates a tape which can't be restored", "always copies every file that is on disk when the dump began") are more desirable than those with weaker guarantees ("always succeeds in favorable circumstances").

Another problem seen too often in the world of backup programs is speed (or lack of it). Some backup programs run as slowly as a few tens of kilobytes per second - certainly too slow in a world of multiple gigabyte disk systems. While affordable 9-track tapes have ranged from 450KB/sec to 750KB/sec in speed, the capacity of 9-track tapes is rapidly falling behind other peripherals, especially in light of the frequency of operator invention required

to change tapes.

Exabyte 8200 drives can achieve sustained throughput of just under 250KB/sec - slower than 9-tracks but with a 2+GB capacity. The new Exabyte 8500 drive runs twice as fast and has double the capacity - thus affording relatively intervention-free backups. Sites with 10GB of storage that desire intervention free backups can probably afford two drives. New compression devices claim compression rations that range from 2x to 5x - thus extending the capacity of a single high-density drive to between 10GB and 25GB (with concomitantly higher transfer rates). Dump should, of course, be able to drive these devices at high speed.

Optical disks and quarter-inch tapes are two other backup media in use - but they're relatively slow when contrasted with 9-track drives. A backup program should be able to exploit 500KB/second drives in order to perform full backups on a 12 GB disk system in eight hours or so (four hours if one has two 8mm drives).

The *dump* program should also keep an catalog of files it backs up. Furthermore, the backup system should include a means to sequence through local and remote partitions (see the section on the 'sequencer', below).

Additionally, the *dump* program should support label verification to reduce operator error and data loss associated with mounting incorrect tapes. Of course, producing tapes that are backward compatible with the old *dump* program would be a definite plus.

Dump should also reduce operator intervention by recognizing and dealing correctly with end-of-media marks. The current *dump* program relies on knowledge about tape length in order to know when to change tapes. A program which correctly recognized end-of-media could, in fact, enable packing many file systems onto a single tape.

Any redesign of the *dump* program should investigate implementation of other 'niceties' that users have requested for many years: mail notification in case of errors, true incremental dumps, and automatic device switchover. Furthermore, some sites request support for "parallel dumps" in which two independent sets (fulls and incrementals) of dumps are interleaved in time (i.e., on alternate days). This insulates a site against loss of data due to dump media failure. Support for this option requires *dump* to be able to use different */etc/dumpdates.** files.

Requirements For Restoring Files

The UNIX *restore*(1) program interface is notoriously poor. While dump tapes have an index preceding their data, tar tapes (and others) must be read completely to learn their contents.

A good UNIX backup scheme will support a catalog that is updated by each dump. Users should be able to peruse the catalog (using the familiar *cd* and *ls* commands) in order to locate versions of backed-up files and request their retrieval. Operators should intervene only to the extent required by security. Of course, the backup catalog must also protect against users seeing filenames they are not normally able to see.

While it is clear that users should be able to see information about files that is normally given by the *ls* or *ls*-algs commands, the question arises: How many versions of a file should a user see? The answer to the question should include enabling the user to:

- See the file system as it existed at any point in time
- See all versions of a given file
- See files back to the most recent level 0 dump
- See files all the way back to the first dump ever

Specifying what to restore (files, directories, an entire hierarchy) should be also simple and intuitive.

Another significant problem with *restore* is its speed. Many systems see a restore speed of eight hours per gigabyte of disk. This is intolerably slow - especially since situations which require large restores are precisely those in which users want to get back online quickly.

Why are *restore* and its brethren so slow? It's those synchronous writes. The operating system likes to make absolutely sure that file systems are not corrupted badly when a machine crashes and all the data stored in its memory are lost. The kernel carefully ensures that data from directory operations (like "create a file") is physically transferred to disk before other operations proceed. It is this synchrony that slows *restore*.

Providing a per-file-system switch to disable this security feature (and increase the probability of totally scrambled disks after a crash) is certainly acceptable in the world of full restores. Since data transfer rates appear to increase by a factor of about four; restarting restores from scratch after the occasional crash still results in no net loss of time.

Requirements For Simple Tape Management

Avoiding the overwriting of tapes is especially important for backups. A simple tape management scheme that utilizes even as simple an approach as the label field in dump headers would be a dramatic improvement. Augmenting it with a small database to keep track of expiration dates would help even more by automating the choice of "which backup tape(s) to use today".

Requirements For Dump Sequencing

Even novice UNIX system administrators regularly create customized programs and procedures for performing dumps. These programs are usually simple shell scripts, awk scripts, or C code.

The scripts sequence the dumps, remind the operator what's going on, and sometimes spread the workload of full dumps throughout the week. Some of the more sophisticated software attempts to perform low level tape management using UNIX. Other administrators use tape drives (or other secondary storage devices) hosted by other machines (e.g., IBM 370 series machines) and rely on their tape management.

The scripts attack a well-defined (and commonly known) set of problems. They attempt to:

- Minimize operator intervention. It is much easier for an operator or administrator to spend ten contiguous minutes setting up a dump than ten minutes spent one minute at a time walking to the machine room, changing a tape, labeling a tape, and re-starting the dump process.
- Sequence through file systems automatically. The *dump* program is not particularly good at this without assistance.
- Sequence through tapes (and tape drives) automatically. Few currently available programs can save files across tapes with correct end-of-media handling.
- Automate (where possible) tape handling and labeling. Overwriting a dump tape is one of the easiest ways to lose data. Good dump schemes attempt to guarantee that some minimal number of recent dumps is saved and that dumpsets have a minimal lifetime before they are recycled.
- Be simple and understandable. Complex schemes - particularly in multi-administrator environments - can be error-prone (in environments where errors are least tolerable).
- Support the 'remote device user' security model so that clients do not require root access on servers in order to use the servers' tape drives.

Some scripts go so far as to attack the problem of choosing tapes to store off-site. This is a difficult problem when, for instance, tapes contain many different dumpsets.

Supplying a program to ease setup and dumping of file systems is desirable for installations of all sizes. New sites (presumably with novice administrators) will benefit from dramatically eased installation and configuration. Sophisticated sites can exploit a good execution system to ensure that their dump requirements are met.

Attempting to create a single program that encompasses all the functionality of the hundreds of scripts in the world is extremely difficult. Nevertheless, it is important to be able to configure the software to emulate the most popular schemes. The richness of configuration options leads to high flexibility - and high complexity. This means that setting up the configurations becomes an error prone or difficult process in itself.

Requirements For Network Support

Of course all the above requirements should work in a networked environment with file systems and tape drives spread throughout a network. Additionally, operators should be able to interact with all the components of an integrated dump system from anywhere within the network.

A Next Step: Backup Copilot

Backup Copilot enhances the *dump*and *restore* programs in addition to providing three components which provide an easy-to-configure, easy-to-run, and reliable scheme for dumping a site's file systems. These components include a configurator, a sequencer, and a catalog.

The sequencer interprets configuration files created by the configurator. It deduces parameters with which to invoke the *dump* program and then iterates those invocations (potentially both locally and remotely) to save a site's files. The sequencer is extremely robust and knows how to sequence dumps in the face of crashes, holidays, and unknown (frequent or occasional) invocations.

The configurator runs once to set up each configuration file and then later may run in edit mode as configurations change (e.g., adding a new disk or a new diskful system). The configurator presents a simple set of choices that administrators easily discern.

Backup Copilot keeps a catalog of dumped files and information about them. The *dump* program updates the catalog; the *recover* program reads the catalog and provides users with a hierarchical view of all backed up files.

The Sequencer

The sequencer runs on a machine which may have several configuration files (one for each set of dumps to be performed in a single group). Each configuration file uses its own */etc/dumpdates.** file. The file systems to be dumped need not be mounted on the machine on which the sequencer runs though network communications must connect the machines. One can run as many sequencers as desired, but one copy of each configuration file concurrently. It may be advantageous for sites owning multiple dump devices (e.g., two Exabyte drives) to split their dumping chores into two dumpsets in order to

maximize use of the drives.

The sequencer not only interprets these configuration files but also interacts with the tape database. The *dump* program itself knows how to verify tape labels and creates new tape labels (using information supplied through command line arguments and potentially through a data file created for this purpose).

The sequencer also keeps complete logs of its activity so that administrators can gain confidence that dumps are being performed as expected. The logs reside in the /var/adm/dumplog directory and include: tape request messages, all dump output, tape usage information, and error information. A typical site will see 5KB-50KB of data logged daily. Each message includes the date, time, machine, and configuration file that was running to generate the log message.

As one would expect, the sequencer performs both full and incremental dumps. The sequencer, the *dump* program, and the file catalog support a switch that specifies true incrementals (''files changed since most recent of any level 0-9 dump'') rather than the standard level 9 incremental: ''any files changed since the most recent of any level 0-8 dump''.

The sequencer supports cyclical dump schemes (using the level numbers of the standard *dump* program). These are typified by a daily sequence like '05555' which means: on Monday (which is to say 'the first day'), do a full dump, then on Tuesday, Wednesday, Thursday, and Friday perform level 5 incrementals. Some file systems in a dumpset may use a '50555' or '55550' cyclical dump in order to spread the full-dump load across different days ('staggering').

A more complex cyclical dump scheme includes one very similar to Epoch's baseline dump: '08888588885888858888'. In this scheme, a full dump is performed only every four weeks (every 20 invocations, that is). Daily incrementals and weekly 'super-incrementals' follow the full dump. Extending the sequence increases the full-dump interval – not necessarily advisable since susceptibility to tape errors and data loss is increased.

The configuration and sequencer programs run without knowledge of the precise frequency that dumps are to be performed. Knowledge of holidays is complex and requires operator intervention. This configuration file design obviates the need to know anything other than 'this program is being run today and should perform the next dumps in sequence'.

Sequencer Configuration File

Figure 1 depicts a sample dump configuration file. Here's how it works:

- Blank lines and comments preceded by '#' are ignored.
- The 'tapelib' parameter names the tape library to use. See the section below on tape libraries.
- The 'dumpmach' tells which machine owns the online catalog of dumped files. This can be the local machine or a remote machine.
- The 'dumpdevs' line has a white-space separated list of tape devices to use. This list should be homogeneous in type (e.g., all 9-track tapes or all Exabyte tapes).
- The 'block' line specifies the block size that *dump* should use.
- The 'tapesup' keyword specifies the default number of tapes to 'plan ahead' for potential

```
tapelib   tapedata
dumpmach  index
dumpdevs  /dev/nrst16 summit:/dev/nrst16
block     64
tapesup   2
notify    kolstad
rdevuser  rdev
longplay

#    level   multiple   days   min available
keep    0        1       15         3
keep    0       26       -1         3
keep    5        1        5         3

mastercycle 00014

fullcycle 00001 -/          >0555555555
fullcycle 00001 -/usr       >5055555555
fullcycle 00001 -/mnt       >5505555555
fullcycle 00001 -/usr2      >5550555555
```

Figure 1: Typical sequencer file

use. This number of tapes will be 'reserved' (for 23 hours) in the tape library each time the sequencer is invoked. The operator is informed which tapes are reserved and hence will be used as the dump session progresses.
- The 'rdevuser' line specifies which user should be used as the login user of choice for the rmt remote tape protocol (thus relieving the server from having to trust clients with root access).
- The 'longplay' specification indicates that tapes will be left in the drive between successive invocations of the sequencer (thus reducing operator intervention even more, since tapes need only be changed every several days instead of daily).

The next sections describes how long to keep dump tapes before they are reused. Each 'keep' line has four fields besides its keyword: the level described by this line, the 'multiple', the minimum number of days to keep this kind of tape tape, and the minimum number of this kind of tape to keep on the shelf before being reused. The 'multiple' works like this: every dump has a 'fullcycle' number (0 for the first time through the list of file systems, 1 for the second, 2 for the third, and so on). When the 'fullcycle' number is a multiple of the number in the second column, then that particular column applies. A multiple of 1 represents every dump; a multiple of 2 represents every other dump, and so on.

Of course every dump tape 'fullcycle number' is a multiple of 1. The sequencer figures the most restrictive set of 'keep' criteria for any given tape (so each 26th tape is kept a long time, even though the 'fullcycle' number is a multiple of 1 in addition to 26).

Finally, the last lines describe not only the scheme to be used for dumping the file systems but also contain state information about current and recent dumps.

The 'mastercycle' line tells how many times the sequencer has completely traversed this particular dumpset. Its initial value is 0. Additionally, the tape database keys expiration dates off both the date and the 'mastercycle' number. It calculates the longest sequence of tapes that can intervene before, say, three copies of a level 5 incremental. Adding this length to the mastercycle number gives an upper bound of a 'safe' expiration cycle for tapes in the database.

Each subsequent line contains information on file systems to be dumped:
- The keyword 'fullcycle'
- The number of times this particular file system has completed a full dump
- A '+', '-', or '*' with the following interpretations: '+' means this file system has been dumped during this cycle; '-' means it hasn't; '*' means a dump was attempted but failed
- The name of the disk device to dump (potentially including a machine name to indicate remote dumps)
- A sequence of digits (including 'x' for true incremental) indicating dump levels
- An indicator ('>') that shows the *next* dump to be performed on this file system; it is advanced at the same time the '-' changes to a '+' in a given master cycle and resets to the beginning of the level sequence before pointing to the newline

The sequencer traverses the dumpset using the configuration information to format *dump* commands and execute them. It updates the configuration file in place by moving the indicator ('>') to show which is the next dump to perform and changing the '-' to a '+' (or '*' in the case of failure).

This sequencing scheme has several interesting properties:
- No knowledge of holidays or other service interruptions is required
- No knowledge of frequency of execution is necessary: the sequencer always does 'the next right thing'
- Crash behavior is quite benign. Sequencing resumes where it left off (potentially dumping a partition twice)
- It is simple to understand and implement
- It is easily extensible to 'parallel dump sets' in which two sets of *dumpdates* files and sequences are used to increase resiliency in the face of tape loss

Configuration

Two different configuration programs create configuration files. The resulting configuration files are not necessarily different; the configurators are differentiated by the complexity of options they present to the user.

The simplest, *dumpconfez*, takes about five minutes to run; requires relatively low sophistication, mostly sets defaults for configuration parameters, and sets up all other required configuration files. This section will concentrate on *dumpconfez*.

The technically demanding *dumpconfig* takes much longer to run (15-30 minutes), requires relatively sophisticated users, and has few defaults.

Any configuration file is easily edited with curses-based editor (*dumped*).

The *dumpconfez* configuration program asks additional explicit questions about:
- File catalog machine
- Dump sequences
- List of people to notify of successes and failures
- Long term drive residency
- Tape blocking factor

- Remote device user
- Expiration criteria
- Parallel dumpsets
- Tape drives to use
- File systems to dump

Once these questions are answered, *dumpconfez* creates both a configuration file for daily backups and a configuration file for the occasional archival backup. It also sets up all other required configuration files (optionally including root's cron entry) and starts the daemons required by the backup system.

Tape Management

Both the *dump* program and the sequencer know how to manipulate named tapes in a rudimentary way. Falling far short of the protection and interchangeability afforded by ANSI labeled tapes, the tape library nevertheless ensures that tapes are not used before their expiration date has elapsed and that tapes are not inadvertently overwritten because of a tape selection or mounting error.

After tapes are initially used (and the labels are written), no maintenance is required.

Disaster Recovery

Once catastrophe strikes, it is important to be able to recover an entire file system hierarchy - even if the catalog of files (with the names of the tapes upon which they are kept) is lost.

To cope with this potential problem, *dumpex* copies tape library information from the database as the last dump each time *dumpex* is invoked. The tape name and file number is mailed to the notification list. Administrators are additionally encouraged to keep the 'last tape written' in a special place (until a new tape replaces it). This way, the tape is easily found.

A recovery program reads the tape and instructs an administrator with the sequence of tapes to load and programs to run to recover the entire database. Once the database is recovered, file recovery is simple.

Security Issues

Security issues for any dump system revolve around access to tapes and tape drives. Access to a tape implies access to any dumped file on that tape, hence invalidating any security notion. Access to tape drives is often a weak point, since dump tapes may be left on the drive after a dump is complete. Obviously, physical access to tapes invalidates security for a site.

Root access is another problem with dump programs which must be able to read every file on the system. The *dump* program only dumps devices for which the real uid and real gid of the invoker have read permission. The *restore* and *recover* programs run setuid root only while making network connections. The restoration of setuid programs follows rules designed to inhibit security problems.

In the catalog of dumped files, viewing of filenames is restricted by file permissions, just like UFS file system. Users only see the catalog for the machine upon which they are executing (thus avoiding impersonation problems).

In the world of network security, Backup Copilot does not use encrypted data, and thus is vulnerable to programs which use the network interface tap. Spoofing (i.e., of the operator monitor) is quite possible. No special user authentication is performed; secure RPC is not used.

Test and Verification Activity

In parallel with the development process, the test group wrote over 60 major tests for the *dump* and *restore* programs. These included: functional tests, reliability tests, stress tests, and Regression tests. On the way to passing all the tests, Backup Copilot went through repair of 60 bugs (many from the original software), including 6 bugs in the underlying kernel.

The tests have run on different configurations for over 3,200 CPU hours and continue to run as this paper is being written. A single test cycle for one hardware configuration runs 28 CPU hours

Backup Copilot has been in production at RMTC since 3/15/91 and has enjoyed a growing number of alpha and beta site successes (with over 35 current beta sites).

Backup Copilot Limitations

Highly secure customers (e.g., C-2 kernels) should not use any remote *dump* program. Customers with less than 1 Gbyte of disk who already have a dump scheme will benefit most from ease of frequent backups and ease of restore. Customers with difficult requirements (e.g., staging dumps to empty partitions for a day, then dumping the partitions) may find Backup Copilot to be too much trouble.

Customers without Sun-4's or SunOS 4.1.1 can not run Backup CoPilot.

Conclusion

Automatic execution and sequencing of dumps coupled with a rudimentary tape management system dramatically eases the operations load of performing dumps. This paper describes a system that can free operators almost entirely from the daily dump burden and considerably ease the burden of file restoration.

While this program is a logical next step in bringing backups into the modern age, it is understood that new media (e.g., optical disk jukeboxes,

digital optical tape jukeboxes, and other peripherals) will enable *dump*s and *restore*s to be completely automated.

Author Information

Rob Kolstad is a senior staff engineer at the SunSoft Rocky Mountain Technology Center in Colorado Springs and served as project leader for the Backup Copilot project. His interests and charter have included developing software that ensures that workstation and smaller-computer storage management paradigms scale to future, larger servers.

Issues in On-line Backup

Steve Shumway - SunSoft, Inc.

ABSTRACT

Administrators of UNIX systems have long been cautioned to run their backups only while in single-user mode. The specific reasons for this practice have often been shrouded in mystery. Administrators have been forced to make trade-offs between system availability and backup reliability without knowing all the facts. Nevertheless, the majority of administrators opt for the high-availability solution of running their backups on-line, on "live" file systems. We present a catalog of many of the dangers of this practice and a method for constructing backup programs that work reliably when run on active file systems.

Backup program functionality can be broken into two phases: a **scan phase**, where the files to be backed up are identified, and a **copy phase** (or **dump phase**), in which the files are copied to output media (e.g., dump tapes). Backup programs are most vulnerable to data loss while executing their scan phases, during which they expect to see a consistent file system image. Other problems can occur if files are deleted, moved, truncated or modified during the dump phase.

Construction of a backup program that runs safely on active file systems requires kernel (i.e., file system) support to "lock" the file system at critical points during the program's execution. Programs that use on-media data for synchronization (such as *dump*(8) and *restore*(8)) must also take defensive measures to ensure that file system modifications during the dump phase do not result in loss of synchronization that might render all or part of the output media unusable.

Introduction

It is the unfortunate reality of systems administration that files do not always hang around as long as we would like. Whether due to hardware failure or user error, files sometimes disappear when not intended. It is for this reason that we do regular backups (you do run regular backups, right?).

Another fact of life is that it is really hard for administrators to keep their users happy. Users want their systems up all the time (or at least when they want to use them!) and want all their files kept around (and recoverable) forever. Unfortunately, most backup programs do not produce reliable results when run on active file systems, dictating some sort of compromise. This paper is going to explore some of the trade-offs between system availability and backup reliability for different backup schemes and present some suggestions as to how to maximize reliability and availability at the same time.

First, we will explore a taxonomy of backup solutions, including standard approaches based on *dump*(8), *tar*(1), and *cpio*(1). Next, we will look at some of the problems associated with performing backups on live file systems using each approach. Finally, we will explore modifications to those approaches that allow one to perform more reliable backups with minimal impact on system availability.

Backup Taxonomy

In order to understand what can go wrong when backup programs are run on active file systems, we must first understand how these programs work. We can categorize backup programs according to the way in which they access the data to be copied and by their execution profile, i.e., the number of passes they make across the file system during execution.

Access Method

The first classification system is based on the way in which a backup program accesses the target file system. Backup programs can be categorized as using either a **device based** (or **inode based**) or **file system based** access method.

In a device based paradigm, the program accesses file system data by reading directly from the underlying device. This method has the advantage of speed but the disadvantages of coding complexity, limited portability, and data inconsistency. The *dump* program supplied with Berkeley-derived UNIX systems uses this access method.

A device based method requires complete knowledge of how data is organized on disk, i.e., what an inode looks like, how inodes are distributed, how data blocks are allocated, etc. Such programs use very low-level system calls, primarily `lseek` and `read`. Potentially, this makes them very fast but restricts their operation to a particular file system type.

Programs using this method are also prone to various data inconsistencies caused by the operating system's buffering of I/O operations. Since the kernel maximizes I/O performance by scheduling operations asynchronously, the underlying device does not necessarily reflect the true state of the file system at any given point in time. This can produce stale data in the output file, leading to anything from corrupt to missing files.

On the other hand, programs that use the file system interface are generally simpler but slower than those using the underlying device. The advantages of this method include reduced complexity, enhanced portability, and better data consistency. The *tar* and *cpio* programs access files through the file system interface.

Rather than carry extensive knowledge of file system structure, such programs are implemented using loops consisting of calls to `opendir`, `readdir`, `stat` (or `lstat`), `open`, and `read`. Programs using this interface are often recursive.

Because file system semantics are fairly well-defined and consistent across multiple platforms, programs that use this interface are more portable than those using the underlying device. Theoretically, one could write a program that was equally adept at backing up both local and remote (via NFS, say) files.

Programs using the file system interface also "see" a more consistent representation of the data because this interface presents a view using the kernel's buffer pool. Changes made to the file system are immediately visible to programs accessing data through the file system interface.

Execution Profile

In the most simplistic sense, backup programs operate in two phases, scanning the file system identifying files to back up then copying those files to backup media. The first task we call the **scan phase** and the second we call the **copy phase** or **dump phase**. In addition to identifying files to copy, the scan phase is usually responsible for obtaining a snapshot of the file system name space. Potentially, the program uses this information during the dump phase; it is certainly required during file recovery.

Backup programs can be categorized by the number of passes they take to implement these phases. Programs such as *tar* or *cpio* are intrinsically single-pass in nature and exhibit **simultaneous scan and dump** behavior. Programs implemented in this manner make a single pass through the file system and immediately copy files that successfully match the criteria for dumping. The obvious benefit of this method is a potential reduction in the amount of I/O. This paradigm is not without its disadvantages, some of which will be explained in a moment.

Programs such as *dump* make multiple passes through the target file system and exhibit **sequential scan and dump** behavior. In its scan phase, *dump* makes a pass across all files in the file system, accomplishing two things: identifying files for dumping (based on whether their modification time is more recent than the time of a previous level's dump) and taking a snapshot of the directory structure. After completing the scan phase, *dump* copies all appropriate files to the output file in inode order, directories first. *Dump* exhibits slightly hybrid behavior in that the file system name space is both collected and dumped in a single step (during Pass three, in which the directories are dumped).

Note that the execution profile is independent of the method used to access the file system. For instance, *dump*, a multiple-pass program, reads from the raw disk but it might just as easily have been written to access data through the file system interface.

Pitfalls of On-line Backup

The primary requirement of any backup program is that it copy files such that they can be retrieved later in the event of loss. Unfortunately, most current offerings do not satisfy this requirement when run on mounted, active file systems. The problems one can encounter depend somewhat on the implementation of the program in question.

File Movement

By far, the most common and serious threat to backup reliability is file movement during the backup. In particular, directory movement at inopportune times can lead to the complete omission of large sections of the file system hierarchy from the backup. Of course, the converse is also possible, in which portions of the file system tree already scanned or copied are revisited. Programs using combined scan and dump algorithms and file system access (such as *tar* or *cpio*) are most vulnerable, but all backup programs face this problem to some extent.

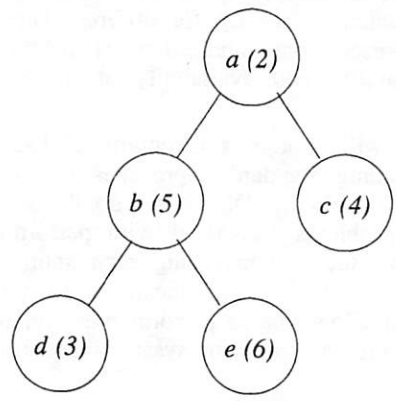


Figure 1: Initial file system state

Figure 1 presents a simple file system tree, with inode numbers shown in parenthesis (we will use them later). For the moment, let us assume that *a*'s directory entries are ordered such that *b* precedes *c*. A program such as *tar* using the file system access method would then traverse this file system in the following order: *a*, *b*, *d*, *e*, then *c*.

Suppose we execute the command

```
% mv /a/b /a/c
```

(the result of which is shown in Figure 2) before our backup program gets a chance to traverse directory *c*. The hierarchy rooted at *b* will be revisited when directory *c*'s sub-directories are traversed. If this occurs during the scan phase, the file will appear to exist in two places in the file system. If this happens during the dump phase, the program's output may very well contain a second copy of the files.

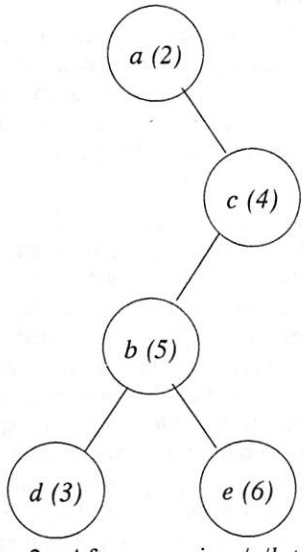


Figure 2: After renaming */a/b* to */a/c/b*

Although, this is relatively benign, it can lead to more serious errors depending on how the program is written. In particular, programs using depth-first file system traversal and returning to the parent directory via the ".." entry can lose their place in the file system if traversing a section of the hierarchy in the process of being moved. Fortunately, *tar* is recursive and uses the stack to avoid this particular situation.

Returning to Figure 1, let us now assume that *a*'s entries are ordered *c*, then *b* and execute the same command. The file system looks the same to the user but the results are disastrously different. If the command is executed just after completing the scan of directory *c*, the backup program will never see directory *b* and the hierarchy beneath it.

If this occurs during the scan phase, the backup program will at best not be able to obtain pathname information for the transported files (it may have a chance to recover during the dump phase, however). At worst, this will lead to the inability to find and copy the file during the dump phase. If the movement occurs during the dump phase, the files will not be present in the program's output. Even worse, subsequent "incremental" backups will believe the hierarchy to have been backed up because the missing files' modification dates will be older than the date of the previous backup. In all likelihood, this insidious error would only be detected during an attempt to restore the files, at which point it is probably too late.

Device based access methods are less prone to problems created by file movement simply because files do not really move. In a UNIX file system, moving a file involves modifying only directory entries, not the mapping between a file and its inode, or the mapping between a file and its data blocks. Thus, a program accessing files through the raw device is immune to file movement once it enters its dump phase. Note that *dump* must complete the pass in which it copies directory blocks to tape (pass three) before reaching this stage, because it combines the functions of scanning and dumping during this pass.

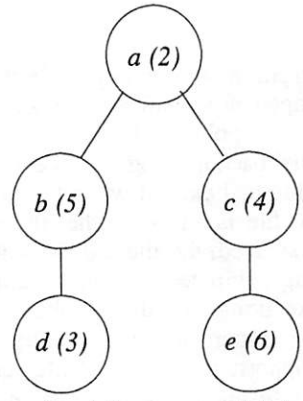


Figure 3: After moving */a/b/e* to */a/c*

Programs like *dump* are still vulnerable to file system name space inconsistencies which can lead to unrestorable files (even though they might be on the tape). In the sample file system shown in Figure 1, a program like *dump* that scans the device in inode order will examine directory *c* (inode 4) before directory *b* (inode 5). If we execute the command

```
% mv /a/b/e /a/c
```

(yielding the result illustrated in Figure 3) after *dump* has scanned directory *c* but before it gets a chance to scan directory *b*, it will still copy *e* during its dump phase. However, the backup tape will contain no path information for *e*, making it unrestorable[1]. The extent of the loss depends on whether *e*

[1] *restore* assumes files without path information were created after *dump* began and discards them with the message "expected next file <x>, got <y>".

is a file or directory and, if a directory, the size of the hierarchy below it.

Conversely, the program might also dump the source directory before the destination directory. In this case, *restore* creates an extraneous hard link (linking the file into both directories) when it restores the file. This is benign but nevertheless incorrect.

The Compress Syndrome

A special case of file movement is what we have nicknamed the "*compress* syndrome". This condition occurs when a program such as *compress*(1) that converts a file from one format to another (removing the original) is run. If the output file is placed in a portion of the file system that has already been scanned or dumped and the input file is removed before the program gets a chance to visit it, or if a file's name is changed (e.g., *file* to *file.Z*) after the scan phase but before the dump phase encounters the file, then neither version of the file will appear in the dumpset (with obvious consequences). This scenario can occur in all programs, regardless of access method or execution profile.

File Creation

The ramifications of creating a file in a file system being dumped are relatively innocuous. If the file is created at a point in the file system not yet traversed by the backup program, the effect is the same as if the file had existed when the program was invoked. If the file is added to the file system at a point already traversed, the file will be skipped if the program is using combined scan and dump, or if the program is executing its dump phase. As with movement, a program using sequential scan and dump has the opportunity to detect files created during its scan phase and copy them during its dump phase. We say this case is relatively innocuous because even if the file is not copied during this backup, it will be copied during the next incremental backup, unlike some of the failure scenarios associated with file movement.

File Deletion

There is very little a backup program can do if a file is deleted before the program reaches that point in the file system. One can only hope the file was saved during the previous backup. Programs using the file system interface could conceivably become "lost" should their current or some set of parent directories be removed out from under them. Again, properly written programs can defend against this condition by avoiding the use of "..".

File Modification

Appending to a file being backed up will probably produce an incomplete representation of the file being updated for several reasons. First, programs such as *dump* obtain the size of each file before copying its data and will only copy this amount of data. Second, reading is almost always faster than writing. This means that a process reading a file that is being updated will eventually catch up with the process writing the file and will see the end of the file. Third, the writing process may be driven interactively, in which case the associated "think time" will accelerate the process of "catching up" just described. If the file update is occurring in place the results are somewhat unpredictable; the backed up file may contain a mixture of old and new data.

File Truncation

In general, if a file is truncated as it is being backed up, the backup program will only be able to copy data up to the point of truncation. Backup programs using *mmap*(2) to copy data must be careful not to attempt to read past this point or be prepared to catch `SIGSEGV` and react accordingly.

A device-based program will not react as quickly to truncation as a program using file system access, which may lead to inclusion of spurious data blocks in the backed-up image of a file. This could occur if data blocks originally allocated to the truncated file are re-used after the program has collected the file's block allocation list but before it actually copies the data in the blocks. This scenario can potentially lead to data corruption and security violations. For example, a programmer might truncate his source file and on restoration discover it contains the corporation's complete salary information!

Another quirk (really of *restore*) can lead to the unusability of large portions of a dump tape. If a file is truncated such that *restore* is presented less data than expected, the truncation boundary must be such that *dump* includes at least 8 Kbytes of data (actually, the maximum file system block size) in the dumpset. Should a file be truncated such that this condition cannot be satisfied, *restore* will immediately abort, rendering all data on the dumpset from that point on unusable. This loss could be catastrophic if this error occurs near the beginning of a full dump.

Solutions

We now examine some of the techniques one can use to build a backup program capable of producing reliable results when run on active file systems. Note that what will be discussed in this section is only necessary on mounted, active file systems. The tasks described here are unnecessary when backing up unmounted, quiescent file system.

Defending Against Movement

File System Locking

One way of ensuring a consistent file system name space during a backup is to employ kernel support to block such operations. Unfortunately, although this solution is extremely reliable, it is also very difficult to implement. It requires kernel source and very sophisticated knowledge of how the UNIX file system works. How long the file system would remained locked depends on the access method used by the program - device based or file system based.

In a device based strategy, one would block `rename` and `unlink`[2] system calls during the scan phase, until the program obtains a consistent copy of the file system's name space. In the *dump* program, this would mean locking the file system during passes one, two, and three. After this point, the program can release the locks as subsequent file movement would not affect the process of locating and copying the remaining data blocks.

A file system based strategy dictates that the locks be held for the duration of the backup, as movement while files are being copied can lead to file loss (as discussed above). A hybrid strategy or position-independent (i.e., inode-based) way of opening files during the dump phase would allow the program to release the locks during its dump phase.

Since a file system is arguably less available while it is locked, the amount of time so spent should be minimized as much as possible. This requirement makes programs based on simultaneous scan and dump algorithms less attractive because they are required to lock the file system for the entire backup. One important point to bear in mind is that although the availability of one file system may suffer slightly, overall system availability is maximized.

Detecting Moved Files

Without kernel support, it is difficult to obtain a consistent backup from an active file system. One possible approach is to detect file movement on the next incremental backup and ensure the moved files are copied then. For instance, a *tar* based scheme might miss an entire hierarchy during a full backup. If the backup program kept a list or database of the files it backed up, the next incremental run could check each file it encountered against the list from the previous backup. If the program discovered a file with a modification time older than the date of the full backup but not in the database, it could assume that it missed the file due to movement and back the file up anyway.

[2]The combination of `link` followed by `unlink` can be equivalent to `rename`, whereas `link` by itself is rather benign.

Defending Against Modification

File System Locking

Again, if changes are to be disallowed during the backup, the best strategy is one based on locking the file system against modifications. System calls such as `write`, `truncate`, `mkdir`, `link`, and `mknod` all cause modifications to files or directories. In addition, files may be modified without the use of explicit system calls if they are mapped into the process' virtual memory (with `mmap`).

Active File Detection and Response

Another approach depends on detecting modifications rather than preventing them. This strategy is based on comparing the file's modification time prior to and after dumping and potentially some period of time after dumping. If the times prior to and immediately after dumping are different, the file was modified as it was being dumped. Comparison of file modification times some time after the file has been completely dumped will indicate whether the file was still in the process of being updated after its dump "completeed". This is the so-called "slow writer problem" and is common on file systems exported to NFS.

Should subsequent activity be detected in this manner, the same file could be "re-dumped" in an attempt to collect a complete version. In order to prevent the output media from being filled by copies of this same file, this is something that cannot be done repeatedly or if the file is very large.

Defending Against Truncation

It should come as no surprise that file system locking is the only way to prevent files from being truncated as they are being backed up. In fact, this problem is really an offshoot of the previous one (modifications) and is solved concurrently. Should locking not be available as a solution, the implementer of the backup program must be careful to code defensively, assuming that a file could be truncated out from under the program at any time.

In particular, if the copy mechanism involves the use of *mmap*, truncation could lead to an attempt to read past the end of the file, which generates a segmentation violation (`SIGSEGV`). This signal is easily caught, but the programmer must be aware of this possibility.

Programs such as *restore* that use on-media data for synchronization can easily lose their place in the stream if *dump* provides them with less data than they expect. The programmer must take care to ensure than under no circumstances does the restoration program lose synchronization, as this effectively results in the loss of data.

A Working Dump Paradigm

Goals

In the development of our on-line solution, we defined the following criteria for what constitutes a "good" dump:

- The file system being dumped remains on-line – most file operations proceed.
- All operations performed by NFS clients on file systems being dumped proceed unless explicitly locked.
- All operations performed by NFS clients on file systems not currently being dumped proceed as usual.
- All files present at the start of the dump are backed up and can later be restored (this is optional).
- If a file changes as it is being backed up, *dump* attempts to copy the most recent version of that file (optional, subject to site-definable restrictions on file size and number of retries).
- *dump* provides notification of changing files if a consistent (and complete) version cannot be dumped (again, optional).
- Files created after the dump begins are dumped no later than the next incremental backup.

Our new *dump* program addresses these criteria through a combination of kernel support (including file system locking and file hole detection), an undocumented position-independent access method, active file detection, and various defensive measures. While designing the system, we recognized that there are inherent trade-offs between some of the goals outlined above and file system availability; therefore, some of the features that ensure completeness are optional. However, our primary goal was that under no circumstances would we produce a dumpset that could not be restored. We make the guarantee that any file on the dumpset can be recovered.

Name Space Consistency

To meet this goal, we added file system locking to the standard UNIX file system. Our implementation includes five lock levels:

- **none** – all operations are allowed to proceed.
- **write** – any attempt to create or modify a file in the target file system is blocked.
- **name** – any attempt to move a file in the target file system is blocked. This blocks `rename`, `rmdir`, and `unlink`.
- **delete** – any attempt to remove a file from the target file system is blocked.
- **hard** – any attempt to access the target file system fails.

A locked operation blocks until that particular lock is released (with the exception of the hard lock, which returns an error). NFS clients attempting locked operations block and behave as if the server is down (with respect to the locked file system, only) generating the familiar "server not responding" message. Operations pending against unlocked file systems are not affected by locks on other file systems.

In our solution, *dump* installs a **name lock** (blocks all `rename`, `rmdir`, and `unlink` operations) during its scan phase until it has dumped all directories in pass three – this takes on the order of 5 minutes for a 1 Gbyte file system. At this point, by default we relax the lock by installing a **delete lock**, preventing removal of files for the duration or the backup.

We recognize that this might be too intrusive for some environments, so we optionally allow the administrator to specify that all locks are to be removed during the remainder of the dump. By the same token, we also realize that some file systems may require complete locking, so we allow the administrator to specify that a file system should be **write locked** for the entire backup.

We take care to avoid deadlocking by refusing to write output files on locked file systems. We also track whether the directory data has been completely committed to the dumpset. This last point is important in case a device error necessitates restarting from a volume checkpoint. If the error occurs on a volume containing any of this dump's directory data, we must restart the dump from scratch because once *dump*begins regular files it relaxes the name lock. Without restarting the dump, we cannot make the guarantee about the dumpset's restorability.

Data Consistency

We wanted to attempt to obtain the most consistent copy of each file's data as possible. To attain this goal, we needed a method of accessing files through the file system in order to use data in the kernel's buffer pool. Traditional access routines such as `open` or `fopen` were not acceptable, as *dump* does not maintain the requisite path information.

We chose to use an undocumented option to `fcntl` that allows the programmer to specify the file to be opened by inode and inode-based generation number. This routine returns a file descriptor as if the file had been opened with `open`, but does so in a completely name-independent way. In conjunction with this mechanism we invented a special `ioctl` call that returns a representation of the file's block allocation list, so we could build the map of holes required by *dump*.

Active Files

A file is deemed "active" if its modification time changes between the time the file is opened for dumping and the time its last byte is copied. Should

*dump*detect it adds the active file to an internal **active file list** and the file become a candidate for redumping.

Whether the file is dumped again depends on site definable criteria such as the file's size, type, and number of previous retries. *Dump* maintains a table of such criteria that specifies under what circumstances files should be recopied. In order to prevent the output media from being filled by copies of a small number of files, this table typically limits the number of retries to some small number (possibly zero) for files that cross certain size thresholds.

Once *dump* completes its dump phase (copying all files to the dumpset), it closes the dumpset (writing two tape marks if the dump volume is a tape) and makes a pass through the active file list. Should an active file match the criteria for redumping, its retry count is updated and it is added to an internal list of files to be recopied. *Dump* generates a status message for each file on the active list informing interested parties of the file's status (either incomplete or attempting to recopy).

If any files are present on the recopy list after completing the scan through the active list, *dump* appends a complete dumpset (it looks a little like an incremental dump) containing the files on the recopy list. *Dump* repeats this process using the previous iteration's recopy list as the basis for the new active list until it produces an empty recopy list (either all files are successfully copied, or the number of retries is exhausted). The dumpsets produced by this method are fully compatible with the current version of *restore*.

Defensive Measures

We take great care to ensure that *restore* never loses synchronization, regardless of what might have been happening to the file system during the dump. In particular, we sense file truncation and generate the amount of data that will keep *restore* in sync. If locking has been released during the dump phase, files can be deleted at inopportune times, such as during volume switches. We also ensure that *dump* generates the appropriate amount of data after a volume switch, even if the file being continued was deleted during the time it took to mount the next tape.

Conclusions

Much can go wrong during an on-line backup if the program in use does not contain appropriate safeguards. File movement is probably the greatest threat to reliability as it can cause large sections of the target file system to be omitted from the backup, render sections of the file system on the backup media unusable, and prevent sections of the hierarchy from being copied during subsequent incremental backups. Unfortunately, taking systems down to perform backups severely impacts availability, so a large proportion of administrators elect to take the risk and perform their backups on-line.

Our new backup scheme attempts to mitigate the problems of system availability and backup reliability through the use of file system locking to ensure a consistent file system name space. Within this constraint, we designed the system to be configurable, leaving some of the trade-offs between availability and completeness to the discretion of the administrator. A solution such as this is not for the faint of heart; it is easy to deadlock either the program or the system if not implemented carefully.

Author Information

Steve Shumway is currently a Software Engineer with SunSoft, Sun Microsystems' newly-created software subsidiary. He works at the Rocky Mountain Technology Center in Colorado Springs developing advanced administrative applications. Before joining Sun, he spent a year with Prisma, an ill-fated start-up (also in Colorado Springs) where he hoped to make big bucks. Prior to entering the industrial ranks, he spent several years at Duke University as the Senior Systems Programmer for the Computer Science Department. Steve has a B.S. in Zoology from Duke and completed the course-work for a Master's degree while working there. Steve is married and enjoys skiing, mountain biking, windsurfing, and volleyball, among other things. He can be reached via electronic mail as shumway@central.sun.com.

A Database for UNIX Backup

Jim Engquist - SunSoft, Inc.

ABSTRACT

When restoring files with existing UNIX backup products, one must peruse the backup media to determine what files were backed up. This makes file restoration cumbersome. We describe a backup database that presents a unified view of all backed up files. This database handles multiple backed up versions of files, centralizes information about files from many different machines, has low overhead on the backup program, and can be updated very quickly. This database allows easy interactive perusing of files for restoration. Tape mounts are not required to select files since all file information is stored online.

The database stores information about backed up files in a hierarchical manner similar to the UFS filesystem. Inode data is available in the database so that filesystem security can be maintained and descriptive information like owner, size and modification time is available for each file. The database can consume between 3% and 8% of typical filesystems that it describes.

Introduction

In many UNIX backup systems, file restoration is a tedious process requiring multiple backup tapes to be located, mounted and scanned before the desired files are located. An online database of backed up files reduces this burden substantially. Files to be restored, and the tapes that contain them, can be identified using the database. Only tapes that contain files of interest need be located and mounted when performing file restoration.

This paper describes an implementation of such a database system. First the requirements for the database are discussed, followed by a description of the implementation. Finally we discuss the performance (in terms of space efficiency and update speed) of the implementation.

The Problem

Our current database implementation is used exclusively to describe backups made with *dump (8)*. At present, there are no provisions for dealing with backups made by other utilities.

A good backup database must provide the following capabilities:

1. Fast insertions of backup tapes
2. Fast deletions of backup tapes
3. Fast and easy traversal
4. Security
5. High Capacity
6. Space efficiency
7. Robustness

Fast Insertions

Database insertions need to be fast and reliable, as we don't wish to slow the backup procedure.

Fast Deletions

The database needs to be able to remove all information associated with a given tape very quickly since "scratching" or re-using a given tape is a common operation.

Fast and Easy Traversal

It should be easy to locate a given file in the database. We chose a model based on the hierarchical file system familiar to UNIX users. One may locate a file in our database using methods very similar to those used to locate a file in the UNIX filesystem.

Security

The database must not allow breaches of file system security. Users must not be able to see names of files contained in protected directories, or data in protected files.

High Capacity

The database must be able to describe backups of large file systems. We planned for multi-gigabyte filesystems comprised of hundreds of thousands of files.

Space Efficiency

The database must not be wasteful of disk space. In particular, the database should be able to describe additional backups of a file it already knows about for a minimal incremental cost in disk space.

Robustness

The database must be robust in the face of system failures. A crash during a dump which updates the database must not corrupt the database.

Third Party Databases

Based on the above requirements, we chose to implement a special purpose database rather than utilizing existing technology like *dbm*(3x) or a commercial relational database. A database such as *dbm*(3x) does not scale well for our purposes (given its restriction that all keys that hash together must fit on a single block). We felt that our requirements for speed (of insertion and deletion) and space efficiency would be particularly hard to meet with a general purpose commercial database. Also, we did not wish to pay database licensing fees per backup site.

Implementation

Structure

The database needs to describe the hierarchy of dumped file systems, along with descriptive data about each dumped file (inode information and containing dump and tape), information about each dump, and information about each tape. The database is implemented by a RPC server process and a hierarchy of data files. In this section we describe each of the data files that comprise the database. For a graphical representation of the files and their relationships, refer to Appendix A.

Note that there are distinct sub-directories for each dumped host. The user of the database views dump entries for a single host at a time.

The database *activetapes* file contains fixed size records that identify all dumps on each tape that is currently "active" in the database. If a given tape contains more dumps than can be described in a single "activetapes" record, multiple records are chained together. There is a single "activetapes" file per backup database.

The database *dir* file contains entries with a structure similar to that of directories in the UNIX filesystem. Each dumped directory is described by one or more fixed size blocks that contain directory entries. If the entries for a given directory don't fit in a single block, multiple blocks are chained together. A directory entry consists of a name, a sub-directory pointer (which is NULL for non-directory files), and an *instance* pointer. Each directory contains entries for "." and ".." to allow standard UNIX style traversal. Note that the *dir* file makes no provision for mount points and file systems. All dumped filesystems from a given host appear as a single seamless hierarchy. There is a single *dir* file for each dumped host.

Because the number of times that a given file appears on backup media is dynamic and cannot be known in advance, the *dir* file uses a scheme reminiscent of indirect blocks in the filesystem to identify individual dumped instances of a file. The *instance* pointers in directory entries are actually offsets into the database *instance* file. The *instance* file is composed of "instance records", each of which contains a fixed sized array of "dump id, dnode offset" pairs. Given this information it is possible to locate information about a dumped instance of a given file using the *dnode* and *header* files described below. Making the instance records fixed size eases management of the *instance* file. If a given file has more dumped instances than can be described in a single record, multiple records are chained together.

The number of instances per record may be configured, but is fixed once the instance file has been created. Making instance records too large leaves portions of the instance file unused (wasting disk space), making them too small means that most instance entries require chained records (with space wasted in "next record" pointers). The rule of thumb is to make the instance records large enough to describe as many full dumps as you plan to keep in your database at any given time. There is a single *instance* file for each dumped host.

Each individual dump in the database is identified by a unique "dump id" and is described by four files: *dnode*, *header*, *symlinks*, and *pathcomponent*. The dump id (actually a time stamp) is appended to each of the four file names.

A *dnode* file contains fixed size records describing each file in the backup. Each record contains *stat*(2) type data (uid, gid, mode, time, size, etc), an indication of the containing tape, an offset into the corresponding *symlinks* file, and some redundant name and hierarchy information to allow recovery of a lost or damaged *dir* file.

Each *header* file contains a single record with information about a dump such as the dumped device and mount point, the time of the dump, the level of the dump, all tapes on which this dump resides and the position (file number) of the dump on its first tape (assuming that it must be the first file on all subsequent tapes).

A *symlinks* file contains the values of all symbolic links in a given dump, stored as null-terminated strings.

A *pathcomponent* file contains each component name of each dumped file, stored as null-terminated strings. Note that the data in this file is redundant (all file name data is available in the *dir* file), and is never used in normal operation. The *pathcomponent* files exist only to allow a *dir* file to be rebuilt in the event of disaster.

Algorithms

Update

Updates of the database are conducted in "batch mode" after a backup completes. This minimizes the impact that database update has on the speed of the dump program and allows for concurrent instances of the dump program. Batching also allows for a robust update scheme which can

recover from system crashes at any point during the update. While performing a backup, the dump program builds a file describing the dumped filesystem, the tapes used, and all files in the dump. When the dump is complete, the database is updated via remote procedure calls. The update operation uses three RPCs:

- begin_update(host)
  1. Generate a unique dumpid
  2. Create a file named "dbroot/host/T.batch_update.dumpid"
  3. Return a file handle which identifies this new file.
- update_data(file_handle, file)
  1. This routine copies data from the given "file" on the client machine to the file identified by "file_handle" on the database machine. Note that XDR is applied to the data as it is copied.
- process_update(file_handle)
  1. Rename "dbroot/host/T.batch_update.dumpid" to "dbroot/host/batch_update.dumpid".
  2. Return the RPC. Now the caller knows that its update file has been successfully received and no retries of the update are necessary. This keeps the RPC short so we don't have to worry about network timeouts. Note that the time it will take to actually apply the update is indeterminate since there may be contention for the database lock.
  3. Lock the database to prevent concurrent updates.
  4. Send an operator message to indicate that an update is beginning.
  5. Scan all existing header files for the given host to ensure that we haven't already processed an update for this dump.
  6. Check the tapes written by this dump. For any tapes on which the first file was written, remove database information currently associated with that tape (if any). See the discussion of the database delete operation for details.
  7. Create a file named "update.inprogress".
  8. Apply the information contained in the update file to the database. This will create new temporary dnode, header, symlinks and pathcomponent files, as well as creating three files of transactions to be applied to activetapes, host/dir and host/instance respectively. This processing proceeds as follows:
     - The dnode, header, symlinks and pathcomponent files are created and written sequentially using stdio functions.
     - The dir and instance files are mmap'ed MAP_PRIVATE in fixed size chunks. We maintain a character array which maps blocks of the file. When a block is modified, we "turn on" the corresponding character in the block map array. When a reference is made to a part of the file outside the mapping, any modified blocks are written to a transaction file and the current mmap'ed chunk is released before mmapping the newly requested chunk. Transaction files are sparse – i.e., when a modified block is written to a transaction file we seek to the block offset and write the block, possibly leaving "holes" in our wake. This allows for the possibility that a block may be referenced multiple times during update. If a request is made for a block which is not in the current mmap'ed chunk, and the block is marked "dirty" in our character map array, we retrieve the modified copy of the block from the transaction file.
     - Write the dirty block maps for the dir and instance files to temporary files.
     - Record this dump in the activetapes file (creating a new activetapes record if this is the first dump on the tape) using the same transaction file and block map strategy used for dir and instance file updates.
  9. Rename "update.inprogress" to "update.done" indicating that the entire batch update file has been processed.
  10. Remove "batch_update.dumpid".
  11. Rename the temporary files (dnode, header, symlinks, and pathcomponent) making them valid database components.
  12. Apply transactions to the dir, instance and activetapes files. Here we again *mmap*(2) the files in fixed size chunks, this time MAP_SHARED. Using the dirty block map created previously, we read modified blocks from the transaction file and copy them into the MAP_SHARED mapping of the original file. When a transaction block is outside the current mapping, we msync() the current mapping, unmap it,

and create a new mapping which contains the desired block.
13. Remove the dir_transaction, dir_map, instance_transaction, instance_map, tape_transaction and tape_map files.
14. Remove "update_done".
15. Unlock the database.
16. Send an operator message indicating that the update is complete.

If there is a crash during the "begin_update" or "update_data" calls described above, the database remains consistent. The caller must retry its update request until it receives a success indication from "process_update". If there is a crash after step 3 of a "process_update" call, the database may be inconsistent. The RPC server will perform the following operations on re-start to recover from a crash during an update operation:

1. Remove any incomplete update files (i.e., files from step 2 of "begin_update" with names like "T.batch_update.dumpid")
2. Check for a "update.inprogress" file. If one exists, we must conclude that a process_update() operation (step 8) did not complete. Thus, we remove all temporary files (including dir and instance transaction files), and re-start (beginning with step 3) process_update() for the "batch_update.dumpid" file (from step 1). If we happened to crash after step 1, but before the end of step 2, the caller may try to re-apply this same update - the check in step 5 will ensure that the database remains consistent in this case.
3. Check for a "update.done" file. If one exists, we conclude that a process_update() failed sometime after step 8. In this case, we re-start the update processing at step 10.

Delete

The database deletion strategy is similar to the update strategy. To remove database information associated with a given tape, the following steps are performed:

1. Lock the database to prevent concurrent updates.
2. Create a temporary update file that contains the label of the tape to be deleted.
3. Locate the activetapes record for the tape.
4. Unlink each of the files (dnode, header, symlinks and pathcomponent) associated with each dumpid in the activetapes record.
5. Locate and remove any references to these dumpids from other tape records. This handles cases of dumps that span tapes. If any part of a dump is removed from the database, the entire dump is removed.
6. Release the tape record(s) in the activetapes file. These modifications to the activetapes file are performed using the strategy discussed in the previous section on database update - using a transaction file and a "block map" file.
7. Close and sync the transaction and map files.
8. Remove the temporary update file (from step 2)
9. Apply the transactions to the activetapes file using the strategy detailed above for the dir and instance files.
10. Remove the map and transaction files.

Database inconsistencies could be introduced by a crash at any time during steps 4 through 9. Each time the database server is started, it will check for the existence of the "update file" from step 2 and re-run the delete operation for the tape label specified in the update file. Note that re-applying a tape delete operation which had already completed does no harm (i.e., the operation is idempotent) unless an update which references the same tape was applied between the delete operations.

Usage

We have implemented a file (and filesystem) recovery program that utilizes this database. It presents a shell-like interface that allows users to traverse the database (and select files for restoration) using familiar commands like "ls", "find", and "cd". Reading the database "dir" file provides data similar to that returned by the *directory* (3V) library routines. Using the instance pointers contained in the "dir" records, one can read the "instance" and "dnode" files to obtain information similar to that returned by the *stat* (2V) system call.

Information for doing restores of full filesystem dumps can be obtained by reading the database "header" files.

Using traditional programs to restore large amounts of data (e.g., sets of full and incremental dumps from tapes that contain multiple dumps) one might need to mount and search a given tape multiple times. A nice feature of our file recovery program is its ability to use database information to order recovery requests. We order requests by file within dump and by dump within tape to ensure that each tape need only be mounted once during the restoration process.

Utility Functions

We have implemented a database utility program which performs a variety of database operations, including:

Tape Listing

A utility function lists database tape information. The available data includes tape labels and descriptions of each dump on a tape.

Deletion

A utility function removes database information associated with a given tape label. In normal usage, tape data will be deleted each time a tape is re-written. This utility is used to remove database data for a tape which will not be re-written (e.g., an error-prone tape).

Addition

There is a utility function to read a dump(8) format backup and add a description of it to the database. There is also a utility to sequentially add information for all dumps on a given tape. These are useful for creating database descriptions for dumps that were not added to the database at the time the dump was performed, or for recovering from disastrous loss of a database.

Compaction

A utility function reclaims unused records in the database. Recall that deletion of database information simply removes the four per-dump files (dnode, header, symlinks, and pathcomponent). This can result in dir and instance records that point "nowhere" (in cases where a given file no longer appears on any of the dumps in the database). This utility function reclaims these dir and instance records so that they can be re-allocated during subsequent database updates.

Server Start/Stop

Utility functions stop the database server and subsequently restart it. The primary use for these is to ensure clean backups of database data. (Note that this functionality could be achieved by killing and restarting the database server process, but this utility provides a cleaner interface).

Database Rebuild

There are utilities to recover various database components in the event of inadvertent deletions or system failures. The activetapes file may be rebuilt from all available header files. Also the dir and instance files may be rebuilt from all available dnode and pathcomponent files. Note that if the per-dump files (dnode, pathcomponent, header, and symlinks) are lost, the database must be restored from backup, or rebuilt by reading all dump tapes.

Performance

Update Speed

Database update speed is effected by the size of the update file, and the amount of time it takes to transmit the file to the database server. The database server can add information to the database at a rate of approximately 2000 dumped files per CPU second.

Space Utilization

The size of the database varies with the number of files dumped, the length of file names, the population of dumped directories, and the number of dumps maintained. A liberal estimate of the amount of space required for the database can be obtained using the formula:

```
space = (220 + 56 * ndumps) * nfiles
                + 10,000,000 bytes
```

The database at our site occupies nearly 600MB. This database describes dumps from two hosts - one of which dumps over 8 GB of disk space. There are roughly 500,000 files dumped from these two hosts. We typically maintain 30 sets of dumps.

Conclusions

The database described here does a good job of meeting the goals we identified at the beginning of this paper. It can be updated quickly, deletions are very fast, it's easy to traverse it in a UNIX file system manner, file system security is not compromised, it's relatively space efficient, and it has high capacity. The only real limit on capacity (other than physical disk space constraints) is the file size limit of 2GB. This means that the number of files which the database can describe is constrained by the size of the dir and instance files. A database with dir and instance files approaching 2GB in size could probably describe about 40 million distinct files (with multiple dumped versions of each).

Acknowledgements

Jeff Polk implemented the tape listing utility. Brian Berliner, Jeff Polk, Joe Cunliffe and Kurt Horton provided invaluable testing assistance and suggestions for functional improvements. Rob Kolstad provided leadership for the project and assistance with this paper.

Author Information

Jim Engquist is a member of the technical staff at Sun's Rocky Mountain Technology Center in Colorado Springs, CO. He is a software development engineer, working in the area of storage management. He holds a BS degree in Computer Science from Iowa State University. Reach him at Sun Microsystems; 5465 Mark Dabling Boulevard; Colorado Springs, CO 80918. Reach him electronically at jde@sun.com.

Appendix A

Consider a database for a root filesystem that resembles Diagram 1.

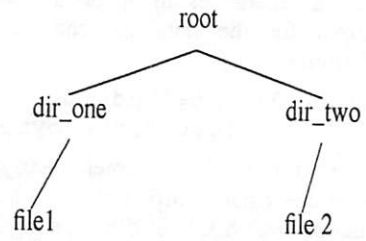


Diagram 1: Example file system

If the filesystem is dumped from a host named "myhost" with an internet address of "129.222.111.222", the hierarchy of files comprising the database for two dumps of this filesystem would be structured as Diagram 2 (note that the symlinks files do not appear since this filesystem does not contain any symlinks, and the pathcomponent files have been omitted for brevity).

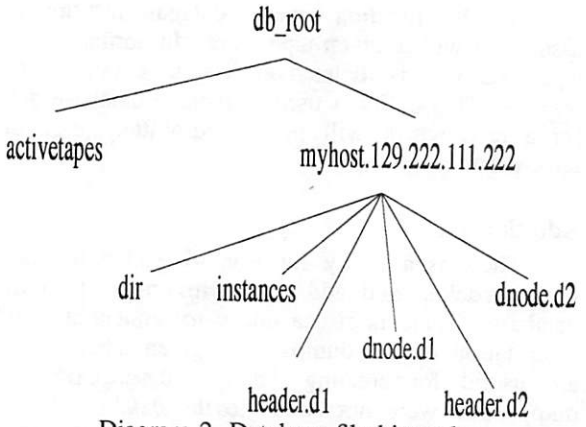
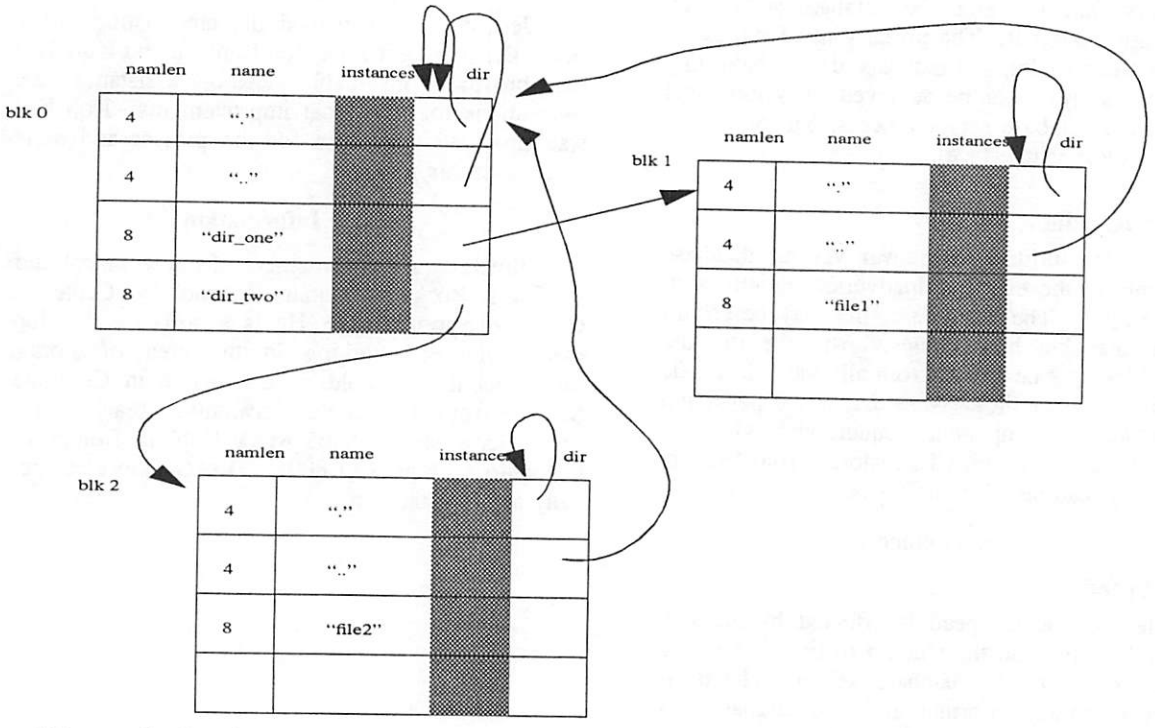


Diagram 2: Database file hierarchy

The dir file for this database resembles Diagram 3. The dir, instance, and dnode files for this database (with the dir pointers omitted for brevity) and their inter-relationships resemble Diagram 4.



ce Diagram 3: Database dir file structure

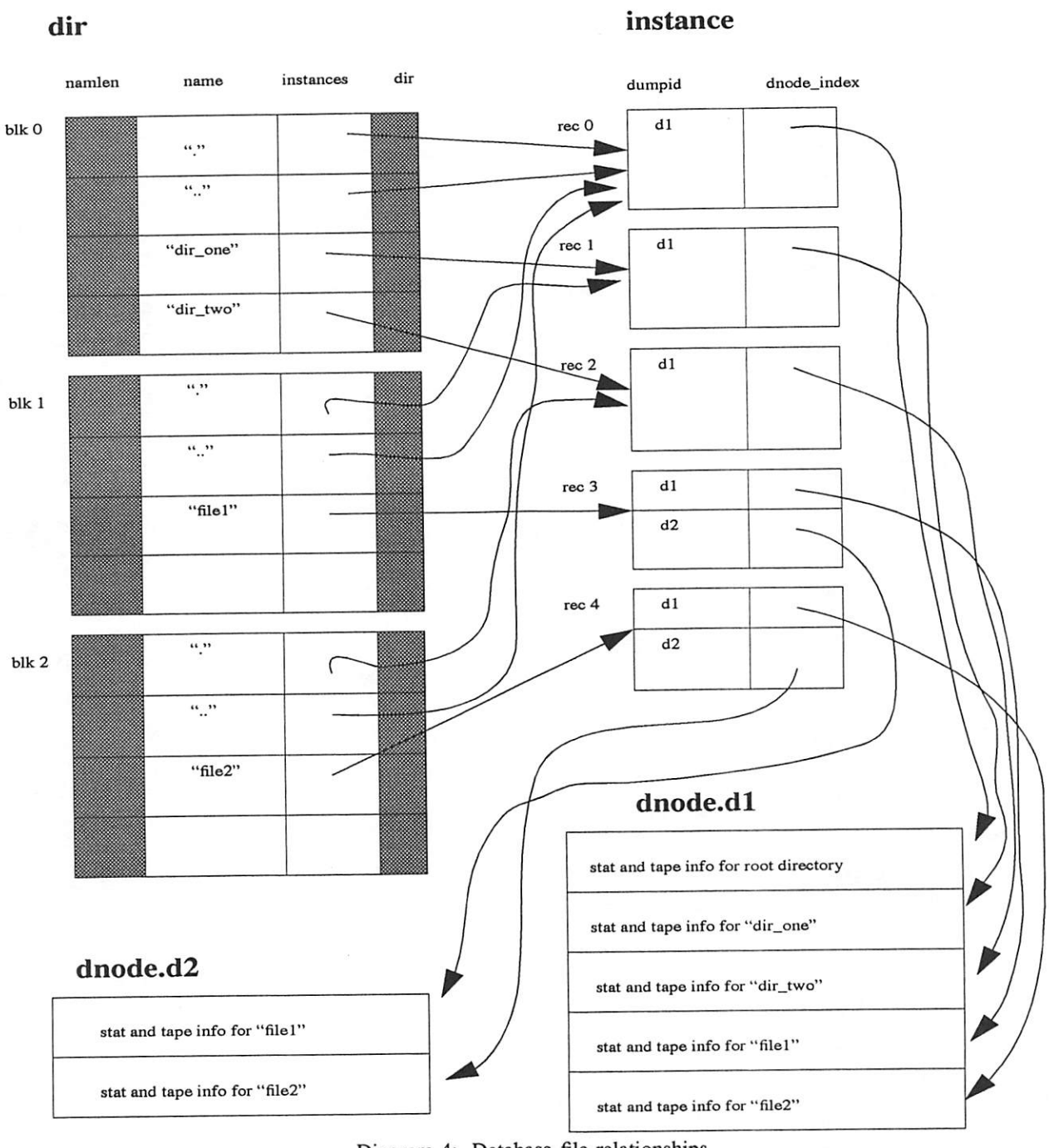


Diagram 4: Database file relationships

A Distributed Operator Interaction System

Steve Shumway - SunSoft, Inc.

ABSTRACT

Several programs used in the administration of UNIX systems require occasional operator attention. A prime example is the *dump*(8) program, which prompts an operator to change tapes. Usually, interactions of this sort can only occur at the terminal from which the program was invoked, making exception handling inconvenient or impossible if the operator has changed location or if the program was invoked via *cron*(8).

To overcome these limitations, we designed and implemented a TCP RPC-based system that enables operators to interact with selected programs, namely *dump* and *restore*(8), from any location on the network. The core technology contains three components: a client library, an operator daemon, and the operator monitor itself.

Client programs (e.g., *dump*) use the library routines to send messages and prompts to an operator daemon, which in turn forwards them around the network to other daemons and any active monitors. The operator daemon is responsible for propagating messages around the network and for acting as the arbitrator among multiple responses to a single message. The monitor is a simple curses-based program that enables its users to view and respond to the messages it receives.

Introduction

Problem Statement

Many of the programs system administrators use in their daily tasks run for long periods of time, occasionally requiring operator intervention. Examples include *dump*(8) and *restore*(8), which copy data to or from tape until that volume is exhausted and then prompt for the next volume.

Traditionally, an operator invoked these programs from a terminal and periodically monitored their progress, satisfying any prompts from the terminal where the program was initiated. This is somewhat inconvenient and can lead to several problems, complicated by the fact that several such programs may be running simultaneously. First, operators are required to monitor the terminal at which the program was started. This tends to tie the operator to specific locations and discourages moving about the facility. It also precludes starting the program from home (via dial-up). Multiple operators cannot conveniently cooperate; they must periodically rendezvous at the point where each program was invoked. Finally, it is difficult (or impossible) to interact with programs started from *cron*(8).

We decided to attack these problems by developing a location-independent system for interacting with programs such as *dump* and *restore*. Our package is implemented as a client library, a system daemon, and a monitor program. Although designed in conjunction with our new backup system, it is general enough that it could eventually be incorporated into a wide variety of applications.

Architectural Overview

The message system is composed of three distinct components: a **status monitor**, a supporting **operator daemon**, and a **client library**. The monitor and the daemon communicate among themselves and client applications to allow operators from around the network to monitor and interact with the various components of our backup system. All communications takes place via TCP-based RPC.

The monitor program (*opermon*) serves two primary purposes: it displays informational messages produced by the various client programs (e.g., *dump* and *restore*) and it acts as the primary interface between operators and those programs when they require human intervention. The monitor provides a simple, *curses*-based interface that enables its users to view and respond to messages from any cursor-addressable terminal (i.e., it is not dependent on the availability of bit-mapped hardware). Users may monitor and control programs incorporating the appropriate support routines (client programs) from any point on the network.

The operator daemon (*rpc.operd*) serves three purposes: to propagate messages throughout the network, to make messages available to monitors as new messages are received, and to arbitrate among multiple responses to the same message. At start-up, the daemon broadcasts its existence in an attempt to prime its cache with messages stored by other daemons around the network. Incoming messages are stored in the cache and forwarded to monitors and other instances of the operator daemon.

Communications Protocol

Message Format

The basic unit of exchange between the components of the package is a **message**. Messages can be merely informative, such as those produced by *dump* to inform operators of its progress, or they can be indicative of problems requiring attention (e.g., tape mount requests). A message is implemented as a **msg_t** structure containing the following information:

- A timestamp in "seconds from the epoch, GMT" format recording the time at which the message was generated and a time-to-live, also in seconds. The two are used together to determine when the message should be expired.
- A **msg_id** structure that uniquely identifies each message. This structure contains the process-ID of the sending process, a sender-generated sequence number, the name of the originating host, and that host's domain.
- The name of the program that generated the message. This is provided for convenience when the message is displayed.
- The name of the host to which responses should be sent. The operator daemon on this host acts as the response **arbitrator**; it acknowledges the first response to a given message and negatively acknowledges any subsequently received responses to the same message. The host assigned the role of arbitrator will usually be that whose operator daemon was the first to receive the message. (i.e., the host specified in the call used to generate the message).
- A **msg_id** structure identifying the target of the message. The target is the message to which a reply or acknowledgement is directed or which is to be cancelled.
- If the message requires a response (i.e., is a prompt), the RPC program number to which responses should be sent, termed the **callback** address.
- Authentication information. This data includes the authentication flavor to be used in the response protocol and the user and primary group-ID of the sending user. Currently, only UNIX authentication is implemented, although we have made provisions for DES authentication.
- A priority level. The level structure exactly matches that of *syslog* (8).
- Type and flag bits. This data controls whether the message is displayed, forwarded to other daemons, broadcast to logged-on operators, and logged via *syslog*. It also marks the message as requiring a reply, constituting a reply, or whether the message is an attempt to cancel or acknowledge (either positively or negatively) another message.
- The length of the message text and the text itself, up to a maximum of 512 characters.

Client Routines

At the lowest level, message transfer between processes is accomplished using four Remote Procedure Calls (RPCs) that implement the functions **login**, **logout**, **sendall**, and **send**.

Login registers a client program with an operator daemon (server). It is only used by programs that need to monitor all message traffic, i.e., *opermon* and *rpc.operd*, but not *dump*. Login calls can either be broadcast to all daemons on a network or directed to a specific daemon. Programs such as *opermon* direct login calls to a single daemon - the one from which they are to receive messages. Operator daemons broadcast an initial login call in an attempt to advertise themselves and establish connections with the other daemons on the network. Upon receipt of a login call, the daemon enters the caller in its **destination list** and will thereafter send any messages it receives to the logged-in program.

Logout unregisters the client program, i.e., it breaks the registration established via the **login** call. Like login, logout can be either broadcast or directed. An implicit logout occurs whenever a daemon senses that a connection to one of its registered client programs has been broken.

Sendall requests that the recipient send its entire message cache to the requester via a series of **send** calls (see below). It is used when a daemon initializes itself in order to prime its message cache.

Send transfers a message from one process to another. It is used by between client programs and operator daemons when messages are generated or responses are delivered, between pairs of operator daemons during message propagation, and between daemons and monitors during distribution of messages and responses.

The above routines are low-level RPC-based primitives and as such are not very easy to use. A more programmer-friendly library that hides the details of RPC programming and message structure sits atop these primitives and contains the following functions:

- `oper_init` - Initializes the message system and attempts to log in to the daemon on the server specified in its argument list.
- `oper_login` - Logs in to the daemon on the server specified in the argument list. This routine may be used to "re-login" to a server or to establish a connection to a different server.
- `oper_logout` - Logs out of the specified server.
- `oper_getall` - Retrieves all messages from the specified server.

- `oper_send` - Creates a message with parameters set according to its various arguments and sends the message to the operator daemon to which the program is connected (established either with `oper_init` or `oper_login`).
- `oper_reply` - Generates a properly formatted reply to the specified target message.
- `oper_cancel` - Cancels the specified target message. Currently, messages can only be cancelled by the process that generated them.
- `oper_receive` - Returns to its caller when either a message is received or when one of the file descriptors passed in as an `fd_set` becomes ready. If a message is received, its sequence number and text are returned to the caller.
- `oper_msg` - An alternative interface for message reception that returns a complete **msg_t** structure to its caller.
- `oper_end` - An optional clean-up routine (optional in the sense of `close` or `exit`).

Operator Daemon

Functionality

The operator daemon is responsible for distributing messages to instances of the status monitor and for arbitrating between responses to those messages. Operator daemons interact with three distinct entities in the performance of their duties: other operator daemons (to propagate messages around the network), client applications such as *dump* and *restore*, and status monitors.

Internally, each operator daemon maintains three distinct lists. The **forward list** is initialized at start-up with the aid of a configuration file. This list controls the retransmission of login calls received by the daemon. When a daemon receives a login call **generated by another daemon**, it will rebroadcast the call to all locally connected networks if attached to more than one **physical** interface, and by direct transmission to any hosts specified in the configuration file via **forward** commands. Either or both retransmission modes may be disabled using the configuration file.

The **destination list** is dynamically adjusted during execution to reflect all programs registered with the daemon as requesting receipt of all messages. The daemon adds an entry to this list upon receipt of a login call and deletes the corresponding entry after receiving either a logout call or sensing problems with the link to the destination program.

The **message cache** contains a copy of every unexpired message the daemon receives (with the exception of cancellation messages). A message is removed from the cache after its time-to-live value is exceeded or it is explicitly cancelled. The cache is implemented as a hash chain (with the hash based on the process-ID and sequence number associated with each message) for quick lookup during cancellation, response arbitration, and duplicate message detection.

The operator daemon responds to four types of events, corresponding to the four RPCs: login calls, logout calls, cache transfer requests, and incoming messages. When *rpc.operd* boots, it broadcasts a login call on the local network informing other daemons that a new daemon has come on-line. Status monitors also use such a (directed) call to bind to a specific operator daemon. Upon receipt of this call, the daemon adds the sender to its destination list. What happens next depends on from where the call originated.

If the call came from another daemon, it may be rebroadcast (if the receiving daemon is connected to more than one physical interface) and forwarded if the forward list is not empty. Additionally, the receiving daemon will attempt a "reverse" login to the sending daemon in an attempt to establish a symmetric connection; messages received by either daemon will be propagated to the other. Should the call come from a user process (such as *opermon*), the receiving daemon will send the contents of its message cache to the new destination via a series of **send** calls.

The entry in the destination list created after receipt of a login call is removed upon receipt of an appropriate logout message. The daemon will also remove an entry if it encounters a broken communications link.

Cache transfer requests will originate from a daemon going through its initialization process. The responding daemon sends each message in its cache to the requesting daemon using the RPC **send** primitive.

Message Propagation

Application processes send messages to operator daemons for redistribution and status monitors send responses to operator daemons for arbitration and action. The daemon will deliver an incoming message marked as forwardable to all programs on its **destination list**. If the message's type indicates it should not be forwarded, it is only sent to non-daemon programs (i.e., monitors). The daemon may perform other actions after receiving a message, depending on the message's type.

In order to receive messages or responses (i.e., either as a monitor or as a program prompting for input), a program registers a remotely callable procedure to which operator daemons can direct **send** RPCs containing messages. This procedure occurs automatically as part of `oper_init`, prior to issuing any **login** or **send** calls.

In this process, the program creates a TCP-based RPC handle, associates an RPC program number in the transient range (0x40000000-0x5fffffff) with it and registers this number and the internal dispatch routine used to actually receive messages with the local *portmapper*(8). A program like *opermon* that is to receive all messages supplies this transient program number to the target daemon as part of the login protocol. Programs prompting for input (such as *dump*) include this number as part of the callback information associated with the prompt message.

This process essentially duplicates the code used by daemons to receive messages. Conceptually, a process receiving a message is acting as a server, even if only for a brief period of time.

Figure 1 depicts the interactions of a sample set of processes on three different systems. Systems **A** and **B** are servers running operator daemons (*rpc.operd*). System **A** is also running a client program, *dump*, and system **C** is running the monitor, *opermon*.

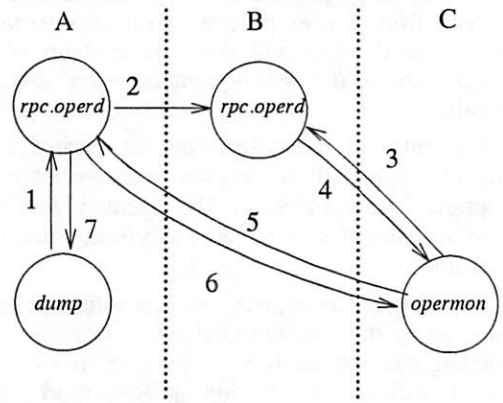


Figure 1: Interprocess communication paths

To retrieve messages from an operator daemon, the monitor goes through the following steps:

1) The originating process (in this case *dump*) sends a message to *rpc.operd* by calling the RPC-based function `oper_send` (which generates the RPC **send** call).
2) The receiving daemon forwards the message to all programs on its internal **destination list**. In this example, the list contains the daemon on host **B**. In order to prevent loops, the daemon on **A** marks the message as not forwardable before sending the message on to the daemon on **B**.
3) A monitor establishes a connection to *rpc.operd* with `oper_login`. In this case, *opermon* connected to the daemon on system **B** but it just as easily could have connected to the one on **A**. As part of the **login** protocol, *opermon* supplies its transient program number to the target daemon.
4) The monitor waits for incoming messages (or command input) using `oper_msg`. A successful **login** call initiates the transfer of whatever messages the target daemon has in its cache and any messages the daemon receives in the future.

Response Arbitration

There are three possible flavors of message response: textual response, cancellation, and acknowledgement (including negative acknowledgement). Textual response involves sending operator input as a reply to a prompt message. Cancellation involves identifying and deleting a previously-issued message. Acknowledgement is performed by the operator daemon acting as arbitrator among a particular message's responses.

When human intervention is required (e.g., to change tapes), programs such as *dump* and *restore* generate a message containing the request for attention. Referring again to Figure 1, if the *dump* process requires a new tape, the message it transmits will request that an operator perform the appropriate tape mount. Such a message requires a response (once the tape has been mounted) before *dump* can continue. Once an operator brings up the monitor, the message is displayed as outlined in steps 1 through 4, above. In addition to its textual component, the message contains information identifying the RPC program the requester (*dump*) has registered to receive a response.

Due to the distributed and asynchronous nature of the message passing protocol, more than one operator may attempt to respond to a given message. The protocol ensures that a process requesting a reply to a message sees only one response and that all responses are acknowledged, either positively or negatively, as appropriate. To accomplish this, each message requiring a response is tagged with the name of the host to which it was first sent; this operator daemon then acts as the **arbitrator** among any received responses.

The remaining steps from Figure 1 illustrate message response and are explained as follows:

5) Send a response (with `oper_reply`) containing the operator's keyboard input to the operator daemon acting as the requesting program's arbitrator. The arbitrator and callback parameters of the incoming message (the one to which we are responding) are used to determine where to send the reply.
6) The arbitrating daemon sends the monitor an acknowledgement, either positive or negative. The monitor receives the acknowledgement and updates its display accordingly. The acknowledgement process and monitor display are detailed below.
7) The positively acknowledged response is forwarded to the program that originally

requested the response.

Each daemon has the responsibility to examine all inbound messages typed as constituting a message reply. If it recognizes a reply for which it is the arbitrator, the daemon examines the target message identifier to determine whether the specified target is present in the local cache and if so, whether a response to that message has already been received.

The arbitrating daemon will generate a positive acknowledgement if the target is present in the cache and an appropriate response has yet to be received. If the message is not in the cache (i.e., the process that generated it has cancelled it) or if a response has already been received, the daemon sends a negative acknowledgement in the same manner. If the responder is not authorized to reply to a particular prompt, the daemon issues a specialized version of negative acknowledgement distinguishing this error from the multiple response case.

The process issuing a prompt is responsible for determining if the responder is authorized to reply. The routine used to receive replies automatically examines the authentication information in the incoming reply and returns acceptance status. In order to be considered valid, the response must have been sent from the super-user, from a member of the group **operator** (on the machine that issued the prompt), or from the owner of the original prompt.

At present, the message system makes the assumption that it is operating in a homogeneous user and group-ID space, i.e., that all systems use the same user and group-ID mappings. For this reason, operation is restricted to within a single domain. Because standard UNIX authentication may be insufficient (i.e., user-ID's might not be considered valid across the network), a future implementation will allow the prompting process to specify that more strict authentication is required.

Processes sending prompts to both the monitor system and their controlling terminal must assume the responsibility for informing the monitor system which source of input satisfied the request. There are two ways in which this situation can be handled.

If input is received from the terminal, the process can send a "dummy" reply message to its arbitrating operator daemon targeting the message containing the prompt. A positive acknowledgement indicates that no monitor-generated responses to the prompt have yet been received; any subsequently received responses will be negatively acknowledged. A negative acknowledgement indicates that the prompt was answered by someone running a monitor and their input was already positively acknowledged. It is up to the requesting program to discard one of the two messages (either the terminal input or the monitor-generated response) and generate the appropriate status messages.

Alternatively, the process can simply cancel the original prompt. Any running monitors will erase the prompt upon receipt of the cancellation request. If this approach is used, the application should flush terminal input so it does not become confused by input in the type-ahead buffer the next time it issues a prompt.

Configuration

A major goal in the design of the login and message propagation protocol was the elimination of configuration files for as many network topologies as possible. We also wanted to avoid dependence on the availability of NIS ("the Yellow Pages"), hence we did not consider using an approach along the lines of the "ypservers" map. We also wanted to avoid using network broadcast wherever possible. In particular, we did not want to use a broadcast-based data distribution method as does *rpc.statd*.

The message protocol has provisions for the establishment of connections across gateways. To take advantage of this feature automatically (i.e., without explicit configuration information) an operator daemon must be running on the machine serving as the gateway between the networks across whose boundary message propagation is desired. If the result of an `ioctl(SIOCGIFCONF)` indicates the presence of more than one network interface, *rpc.operd* will automatically retransmit login calls received on one network to all other connected networks via network broadcast.

Should it be impossible to run the daemon on the gateway, some dependency on configuration files is necessary. One host on either side of the network gateway should be designated as the "message gateway" and contain a **forward** entry in its configuration file pointing to its remote counterpart.

To prevent loops, login calls are only retransmitted the first time they are received. Once a login call is passed across a gateway, the protocol behaves as if the two communicating hosts were on the same local network.

For greatest robustness, a site could run *rpc.operd* on any host that is to run any of the client programs. This ensures that operators can monitor and control the progress of the program from anywhere on the network without depending on any particular machine being up (other than the operator's machine and the machine being monitored, obviously).

At a minimum, at least one operator daemon should be run on each network segment or subnet. Ideally, each gateway machine (between local network segments) will run *rpc.operd*. If this is the case, no configuration files are needed; the system will propagate messages between network segments "out of the box". Note that two network segments could be served by a single operator daemon if that

daemon were running on the gateway.

Monitor Interface

The status monitor, *opermon*, is the operator's interface to the message system. With it, an operator can monitor the status of all client programs running on the network and interact with those programs should they require attention.

The terminal screen is managed by *curses*(3) and is divided into two areas: a message display area and a command area. Figure 2 presents a typical monitor screen. The top of the screen indicates from which server the monitor is obtaining messages and the current time of day. The message display area is surrounded by status lines that are displayed in the event that displayable messages extend either above or below the displayed "window". The command area is delimited by a prompt line at the top and a filter status line at the bottom.

The text in the command prompt adjusts to suit the situation; the "-" and "+" indicators are only present when messages are above and below the window, respectively. The prompt also includes the tags of messages to which the user may respond. The two lines between the prompt and the filter status are used to display miscellaneous status and error information.

Messages are printed in the scrollable display area and remain there until they are cancelled by the originating process or until they expire. Those requiring responses are distinguished from non-interactive (i.e., status) messages through the use of single-character identification tags and dispositions. A typical message requiring a response is a tape mount request.

Commands available to the user from within the command area are grouped into screen commands, local monitor commands, and message responses. Screen commands include scroll up, scroll down, redraw, and suspend screen update. Monitor commands are accessible through "command mode", entered by typing a colon, and include help, connect to server, erase message, filter, and quit. All commands are implemented with "hot-keys", allowing the user to invoke a command with only one or two keystrokes.

The server connection command is useful in the event the server to which the monitor is connected goes down, or if the user wishes to monitor message traffic passing through a different server than the one to which the monitor is connected. Note that in a normal environment, all servers on a given network will "see" the same messages, so this command is useful primarily in the event of server crashes.

The filter command is provided for the convenience of administrators of large networks, where message traffic is anticipated to be heavy enough that it might be difficult to pick out messages of interest. With this command, the user can limit the display of messages to those containing specified regular expressions. After invoking the command, the user enters one or more *ed*(1)-like regular expressions, separated by vertical bars, as in *egrep*(1). Typical expressions a user might enter include program names (e.g., *dump*), host names, or a combination of both (e.g., *dump@rmtc*).

As stated earlier, the monitor tags messages requiring responses with an identifier and a disposition. The disposition follows the tag and indicates whether the user has attempted a response to the message and whether the response was accepted or rejected. Users select a message for response by typing its single character tag (e.g., the letter "a" for the prompt in Figure 2). Once selected, the message is highlighted in reverse video until the reply is completed (by entering a carriage return) or cancelled (by typing the line kill character). All input up to and including the terminating carriage return is sent to the prompting process.

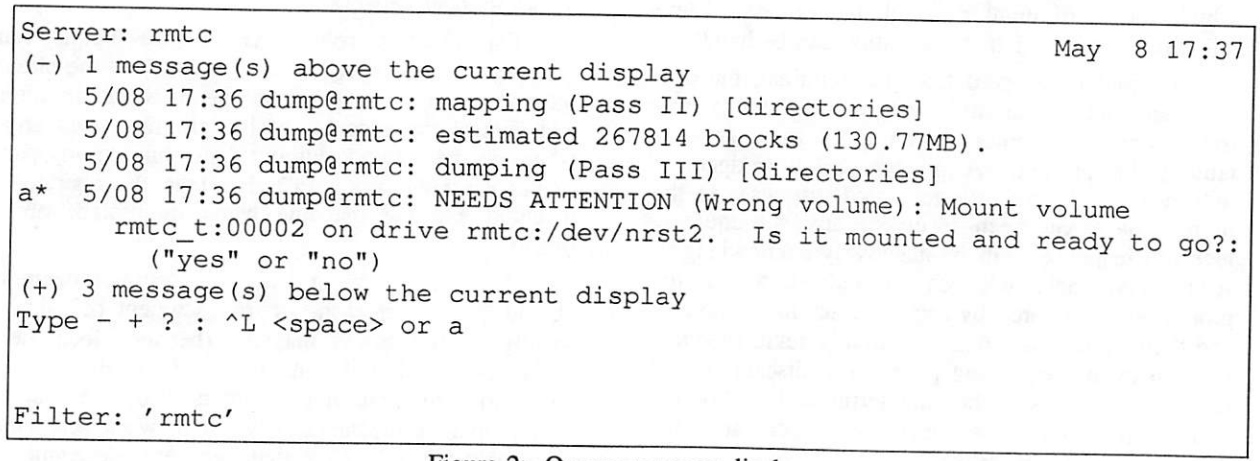


Figure 2: Opermon screen display

Conclusions

Our first users of the package have found the message system extremely useful, particularly when running *dump* from *cron*. In addition to the normal errors that can occur when running *dump*, our package introduces several new error conditions as a result of our new tape label verification feature. Several users have experienced the situation in which the wrong tape was mounted, easily noticed and corrected by using *opermon*.

Although we designed the message system in conjunction with a backup package, we designed it with more generic applications in mind. The specification for the system has not yet been published; however, future releases may include public library support allowing users to plug in their own applications. Other futures include the addition of DES-based authentication, a bit-mapped version of *opermon* with scroll bars and point-and-click message selection, and the ability to display messages that have expired (similar to the history mechanism of *xterm* (1)).

Author Information

Steve Shumway is currently a Software Engineer with Sunsoft, Sun Microsystems' newly-created software subsidiary. He works at the Rocky Mountain Technology Center in Colorado Springs developing advanced administrative applications. Before joining Sun, he spent a year with Prisma, an ill-fated start-up (also in Colorado Springs) where he hoped to make big bucks. Prior to entering the industrial ranks, he spent several years at Duke University as the Senior Systems Programmer for the Computer Science Department. Steve has a B.S. in Zoology from Duke and completed the course-work for a Master's degree in Computer Science while working there. Steve is married and enjoys skiing, mountain biking, windsurfing, and volleyball, among other things. He can be reached electronically as shumway@central.sun.com.

A Flexible File System Cleanup Utility

J Greely - The Ohio State University

ABSTRACT

In 1989, Elizabeth Zwicky described the quota-less disk space management system in use by the Computer Science Department at The Ohio State University[1]. One component of that system was a program to scan a file system and delete "safe" files. This paper describes a new implementation of that utility, which can be configured to meet the needs of many different environments.

Why a Cleanup Program?

When a file system fills and an angry user cries "I can't save my COBOL program," the correct answer is not "That's a feature." For most of us, it means it's time to pull on the hip-waders, grab a machete, and try to free up some space. There are several problems with this approach: it doesn't scale well, the rules for deletion are often poorly-defined and inconsistently applied, and it consumes a large amount of valuable staff time.

Our original cleanup program was based on experience gained cleaning up file systems in our environment, and contained hard-coded rules for dealing with core files, Emacs auto-save and backup files, TeX output files, and object files. Tests were applied (age, age relative to the "parent" file, and difference from the parent file), and matching files were deleted.

Four years later, our environment looks very different, and the old program just isn't effective anymore. We have updated it several times, but it can rarely free up more than a megabyte on a 350 megabyte file system, and more and more we found ourselves spending time deleting files by hand again.

At this point, I started prototyping a replacement in perl[2], with easy configuration as a primary goal. The first time I tested the prototype with a set of rules based on recent experience, it reported more than ten megabytes of deletable files (three percent of the space in use on the test file system). Pleased but paranoid, I carefully examined the output for errors. Finding none, I turned it loose on all of our user filesystems, freeing up almost 120 megabytes (out of seven gigabytes).

Over the past few months I've tinkered with more refined configurations, the most recent of which freed an additional 150 megabytes. As a bonus, the flexibility lets me create custom configurations for specific situations (e.g., a special end-of-quarter cleanup for student accounts). The net result is more disk space with less work. The users are happier, I'm happier, and my manager is calmer.

Ok, Why *This* Cleanup Program?

Cleaning up file systems is one of many "small problems" in system administration. A lot of people confront it every day, but the solutions they come up with are either limited in scope or filled with non-portable assumptions. The result is that administrators are constantly re-inventing the wheel, and not all of their wheels roll. Designing a tool that can be easily tailored to fit into many different environments with different policies reduces the need for sysadmins to "roll their own."

Design and Implementation

Cleanup works by traversing a file system, looking for deletable files. The basic idea is that by examining a directory, you can usually infer relationships between files by looking at their names and modification dates. On a UNIX system, the presence of both `foo.c` and `foo.o` usually means that `foo.o` was created from (or is a *child* of) `foo.c`. If you have a large number of users compiling programs, deleting old, out-of-date, or re-creatable `.o` files may free up a significant amount of disk space. Similarly, frequent text editing will result in a lot of editor backup files (e.g., `foo.c~`), most of which aren't needed after a few days. The system administrator collects rules such as these into a configuration file, which is read at runtime.

Perl's regular expression and string support made it easy to specify parent-child relationships. A parent is a regular expression with sub-expressions (e.g., `/^(.+)\.c$/` for C source files), and children are strings containing references to the sub-expressions (`$1.o` for object files, for example). Each file in a directory is tested against the parent expressions, and if a match is found, the child strings are evaluated. If a child exists, it is examined to see if it can safely be deleted, based on a list of conditions of the form *test+age*, where *test* is the name of a function that checks a specific condition, and *age* is the age in days, passed to the test function. A common condition in my configurations is `older+0`, which indicates that the child should be deleted if it is at least 0 days older than the parent. So, a complete rule for deleting object files from C

programs might be the following:

```
/^(.+)\.c$/ {
  $1.o    older+0|old+45
}
```

Braces are used to allow for multiple children from a single parent, and a vertical bar is used to allow multiple deletion conditions for a child (in this case, the second rule indicates that the child can also be deleted if it has not been modified for at least 45 days).

More sophisticated configurations are possible. *Cleanup* currently supports directory-specific rules, "orphan" rules (e.g., core files, which have no parent), user-definable actions (compression instead of deletion, for example), and user-definable file tests (the default configuration contains tests `same` and `younger`, in addition to the two shown above). The test and action functions are kept in a separate source file, making them easy to enhance or replace.

File tests can do more than just look at modification times. Discussions in the newsgroup `alt.comp.acad-freedom.talk` led me to create a test called `raster`, which reads the first few bytes of a file to determine if it is a (possibly compressed) Sun raster or GIF image. I've only used it with deletion turned off[1], but the results were interesting. Policy prevents us from including this test in any ordinary cleanup configuration, but some other sites may find it (or other magic-cookie tests) useful.

What Do You Delete?

The hard part about making *Cleanup* useful is writing the configuration file. I've tried to simplify the task by providing samples with the distribution, but my opinion of what can be deleted may be quite different from someone else's. A good example is the above rule for deleting object files from C programs.

It seemed quite obvious to me that the appropriate thing to do was delete any object file that was older than the corresponding source file, since it was out-of-date, and would be need to be recompiled anyway. The people on the other side of the dinner table thought that it was better to delete object files that were younger than the source files, since they *could* be remade. Thinking it over, I'm half-convinced that *all* object files should be deleted, but I'm willing to make exceptions.

The moral of this story is that by keeping the policy separate from the tool, two groups with a significant difference of opinion on which files to delete can use the same program, changing only a few lines in a configuration file.

[1] Contrary to what some die-hard flamers believe, we are *not* fascists.

Building a sample configuration

To illustrate the process of developing a cleanup configuration, I borrowed an empty disk for a few days and restored a typical user file system onto it. The following table contains a partial summary of the contents of the file system, categorized by name and extension. The complete table is eight pages long, so I've extracted the most useful parts.

File Category	Total Size (Bytes)	Total Number	Average Age (Days)
TOTAL	358766592	28049	130
`*.out`	92973139	685	37
`*~`	7785292	1574	34
`*.c`	7523595	1268	595
`*.o`	6824702	1412	17
`*.stb`	4612096	129	21
`*.Z`	4010335	104	227
`*.mod`	3597496	749	23
`*.cob`	3248910	370	45
`#*#`	2722217	475	24
`*.p`	2654328	246	49

Table 1

Examination of files in the "*.out" category suggest that they are primarily COBOL output files from student assignments (as are the "*.stb" files). None of the assignments lasts two weeks, so intermediate and output files can be deleted after fourteen days, freeing as much as 29% of the space in use. Another 3% can be gotten by removing GNU Emacs auto-save and backup files ("#*#" and "*~"), for a total possible savings of 32%.

I applied the following cleanup configuration to the test file system:

```
{
  /^(.+)$/ {
     $1~       same+1|old+14
     #$1#      older+0|same|old+1
  }
  /^(.+)\.cob$/ {
    $1.o       old+14|older+0
    $1.out     old+14|older+0
    $1.stb     old+14|older+0
  }
}
```

In just over six minutes, this simple *Cleanup* configuration freed 47 megabytes of space (14% of the space in use). Table 2 summarizes the change in the affected categories.

Using a very simple configuration file, I recovered a substantial amount of disk space. Further savings are obviously possible by adding rules based on the full data from which Table 1 is excerpted. From experience, I'd say that at least another 20 megabytes could be recovered on this file system (and, in fact, *was* recovered by my normal

cleanup configuration), but the amount depends a great deal on usage patterns. My example file system was used almost entirely by undergraduate students, whose use is very predictable. Had I chosen one filled with faculty projects, the results would have been quite disappointing.

File Category	Removed (Bytes)	Savings (Percent)
*.out	36519937	10.2
*~	4191211	1.2
*.stb	3134464	0.9
#*#	2295324	0.6
*.o	1058044	0.3

Table 2

Conclusions

1. Automated file system cleanup frees administrators to handle real problems, and improves the overall level of service.
2. The usefulness of a system administration tool is inversely proportional to the amount of system and environment-specific information contained in it.
3. Perl is a great language for prototyping and implementing system tools, and all sorts of other things, for that matter. For a small cost in performance, you gain a great deal of power and flexibility. (this isn't a *recent* conclusion, but I thought I'd mention it)
4. Ken Thompson wasn't kidding when he said (according to the BSD fortune file), "The steady state of disks is full."

Longer Sample Configuration

Figure 1 shows a longer sample configuration.

Availability

This paper and the software it describes are available for anonymous ftp and uucp on archive.cis.ohio-state.edu and osu-cis, respectively. Please send e-mail to staff@cis.ohio-state.edu if you have difficulty accessing these services.

Author Information

J Greely is a systems programmer for the CIS department at The Ohio State University. He is occasionally working towards some kind of degree in whatever strikes his fancy. His main professional

```
#compress old mail.
#
m|/mail$|i {
  !/.Z$/    old+30   compress
}
{
  #emacs backups and autosaves
  #
  /^(.+)$/ {
     $1~      same+1|older+7|old+45
     #$1#     older+0|same|old+1
  }
  #compress preserves mtime, so we can
  #clean up compressed build areas.
  #
  /^(.+).c(.Z)?$/ {
     $1.o     older+0|old+45
     $1.o.Z   older+0|old+45
  }
  .article   old+7
  .letter    old+7
  core       old+7
  #I don't really use this rule,
  #but isn't it fun?
  #
  /^.+$/     raster+0
}
```

Figure 1: Longer sample configuration

interests are in computer security and simplifying system configuration and administration tasks. Reach him by e-mail at jgreely@cis.ohio-state.edu, and by U.S. Mail at Dept. of Computer and Information Science, 2036 Neil Avenue Mall, Columbus, OH 43210.

References

[1] Elizabeth Zwicky. "Disk Space Management Without Quotas," In *Proceedings of the USENIX Large Installation Systems Administration III Workshop*, USENIX Assn., 1989.

[2] Larry Wall and Randal L. Schwartz. *Programming perl*. O'Reilly & Associates, Inc., 1990.

Fdist: A Domain Based File Distribution System for a Heterogeneous Environment

Bjorn Satdeva - /sys/admin, inc.
Paul M. Moriarty - MIPS Computer Systems, Inc.

ABSTRACT

Administration of a large UNIX site is by no means easy. The tools provided by standard UNIX implementations are, at best, inadequate. The explosive growth in network sizes over the last few years has resulted in larger and more complex sites but no significant new tools to help system administrators maintain these sites; particularly in networked UNIX environments.

One major problem that has not been fully solved is that of automatic file distribution. The *rdist*(1) command works well in small homogeneous environments. However in larger and more heterogeneous environments, it becomes difficult to maintain the rdist command files in an orderly and systematic manner.

This paper describes the design, implementation and practical experience with *fdist*, a domain oriented distribution tool that enables easy management of automatic file distribution in a large heterogeneous environment. *Fdist* is written in perl and uses a slightly modified version of rdist as its underlying file distribution tool.

Introduction

This paper describes a software package, *fdist*, which was developed by MIPS Computer Systems, Inc. *Fdist* provides automatic distribution of files across a large number of heterogeneous systems in a manner which appears virtually homogeneous.

Prior to the development of the *fdist* system, MIPS used a simplistic automatic file distribution approach which was originally implemented to support a much smaller host base. The MIPS scheme also used the rdist command, but the rdist distribution files (distfiles) were maintained entirely by hand. By the time the *fdist* project was initiated, the MIPS corporation had grown to the size where the manual maintenance process of the distfiles had become cumbersome, time consuming and error prone. Some of the problems encountered included:

- The distfile was originally written to handle approximately 10 hosts. As the number of hosts increased, the distfile and the recipient hosts list became a hodge-podge of rules for the various types of hosts. As a result of the lack of a systematic approach to their design, the distfile and recipient hosts list became very difficult to understand and maintain in a reasonable manner.
- Although there were approximately 500 UNIX hosts in */etc/hosts*, only 123 were receiving updates because the remaining hosts were too different to group in the distfile (e.g., they were a hardware platform or version of the operating system that wasn't already modeled). The differences in configuration among hosts was a source of network reliability problems.
- Under sub-optimal conditions, rdist would take more than 12 hours to complete (sometimes due to network load, hosts which were down or unreachable, or especially large numbers of files to distribute).
- Circumstances often arose which necessitated the frequent (more than daily) distribution of some kinds of information (e.g., when adding a new host or changing/updating mail aliases). This wasted considerable network bandwidth.
- Customizing distributions for a single host or collection of hosts was cumbersome. The existing set of rules had evolved into an "all or nothing" scheme.

It became clear that this approach would be unable to handle the expected future growth in the number of file servers, let alone providing similar support of workstations. Therefore, the system administration staff decided to design and implement a system which would be able to handle automatic file distribution of a large number of heterogeneous hosts.

Description

Fdist is essentially a distribution system built on top of rdist. One host, called the distribution master, distributes both the source files and the fdist utilities to five distribution slaves. From the perspective of *fdist*, the distribution slaves are clones of the

distribution master in that each contains a copy of all files which are to be distributed as well as the *fdist* utilities. The distribution slaves then distribute the source files to their respective clients (approximately 100 per slave). Additionally, *fdist* generates and maintains the */etc/hosts* and */etc/ethers* files for all hosts under its control. *Fdist* sends e-mail to notify the system administrator(s) responsible for maintaining the fdist system of any problems relating to the distribution of files. The person tasked with maintaining the *fdist* system is more commonly referred to as the **distmaster**.

Design Goals

The above problems motivated the decision to implement a new distribution scheme. Some of the more important goals for the new *fdist* distribution scheme were:

- Full support for heterogeneity.
- The ability to support a minimum of 1000 hosts.
- The ability to specify distribution based primarily on *use* of the target system rather than hardware vendor and OS type.
- Full support for separating policy strategies from the daily maintenance of the network.
- Centralized control of distribution.
- Ease of use. *Fdist* should enable a less skilled system operator to maintain many of aspects of the distribution without the need to know rdist syntax or any other complex language.
- Enable a simple scheme where files are distributed with an interval relevant to their type and importance. To keep network usage to a minimum it was important to ensure that on any given hour, only a few files flowed through the network instead of the old "distribute everything to the entire world" approach.
- Support good interaction with other tools. We wanted a shell command line approach, rather than a graphical user interface, to ensure that we could integrate the new distribution system with existing tools where necessary. If the need for a menu driven system, or a full graphical user interface arose later, it could always be built on top of the shell command line interface.

In order to achieve these goals, it was necessary to have the ability to specify to which hosts a given file should be distributed, without having to specify a list of hosts on a command line. The approach that was chosen was one which had been discussed in the IEEE POSIX 1003.7 system administration committee which groups hosts into administrative domains. Since the committee gave no suggestions as to what an administrative domain was, the Internet's use of fully qualified, domain style names for e-mail was emulated for the syntax and semantics used in the administrative domain naming scheme. However, the administrative domains are, in this implementation, completely separate from the Internet e-mail domains, and share only the syntax and semantics of the Internet approach. Therefore, each host is part of both an Internet and administrative domain, with neither type imposing any restriction on the other. For the purposes of this paper, the term domain refers to administrative domains (described below).

Administrative Domains

As mentioned above, the administrative domain's syntax was borrowed from the Internet domain structure. However, the semantics are somewhat different and less general. When the domain structures were designed, it was necessary to ensure that:

- Each host could be member of several administrative domains, where each domain is a "view" of a specific administrative problem.
- Hosts could be grouped according to their usage.
- Different hardware platforms and operating systems could be supported.
- Wildcards could be used in place of any domain name component. This was implemented using the domain name **common**, which matches any domain name component. (**common** was chosen rather than the asterisk ('*') character to avoid the conflict with the shell, as it was envisioned that the system would used by less skilled system operators.)

A five component domain name permitted sufficient flexibility to meet the design goals. (*Fdist* should handle an arbitrary number of domain name components but only five were used in this implementation.) Each level has a specific meaning:

- The top level component of the domain name is used for the general specification of the domain. Today, there are only two top level domains in use, *adm* and *prt*, however the *fdist* system will support any number of top level domains.
- The second level component of the domain name specifies the group to which the host belongs. While there are a number of these domains, this paper will exemplify only the cc (computer center) and osgrp (operating system development group).
- The third level component of the domain name specifies the specific operating system that the host is running. In all instances it is necessary to differentiate between SunOS and RISC/os, but in many cases it is also necessary to differentiate between different releases of the same operating system.
- The fourth level component of the domain name is used to specify the hardware platform

on which the OS is running. These are typically real differences between different architectures, like Sun 3 and Sun 4. In some cases, the fourth level distinguishes among system configurations, like the difference between a file server and a workstation.

- The fifth level component of the domain name has not been used much in practice, however when the system was designed it was believed that there would be a great need to distinguish between diskless and disked systems. This has not proven to be the case. Therefore the implementation of *fdist* has been changed to default to the "disked" domain unless diskless is specified explicitly.

Finally, there had to be a way to identify a specific file within a domain. The colon (':') was chosen to combine the domain name with the file name, as this notation has commonly been used to specify a file's path on a specific host. The domain and path combination is called a **domainpath**.

Using the above definitions, */etc/hosts.equiv* file on all disked Sun 4's that run SunOS and are used by the Computer Center can be referred to as:

```
disked.sun4.sunos.cc.adm:/etc/hosts.equiv
```

and the services file on all MIPS workstations as:

```
mipsws.common.common.adm:/etc/services
```

Database structure

This section outlines the implementation of the database structures. The term database is used very loosely, as *fdist*'s database includes a copy of all files to be distributed as well as number of configuration tables, which at this time all are flat text files. These files were supposed to be implemented using a commercial database product, but dependencies on another corporate project have so far not permitted this.

Hostdata File

The **hostdata** file is a text file that contains all the *per host* information. Under the *fdist* system, **hostdata** contains *per host* information, one host entry per line using white space as field separators. This file contains traditional host information, such as ethernet and IP addresses, primary host names and aliases (the latter comma separated). The file also contains the names of the administrative domains that a host belongs to. This information is kept in five additional fields, where the first is a comma separated list of all top level component of the domain name followed by the remaining four domain component levels.

Several "place holders" occupy (sub)fields for which no information available. Only one such name should be sufficient for the computerized part of the system (and in fact is, as all of these names are internally represented as "none"). However, the human interface to the system requires that the people maintaining the system be able to differentiate between several values. Currently implemented values are:

- NONE: No value is used for this field.
- UNKNOWN: There should be a value for this field, but it is not known.
- IGNORE: This domain entry is ignored. This is used in the top level domain component field for gateways, PC's and other non-UNIX platforms, where the distribution information is irrelevant.

The distinction between NONE and UNKNOWN was very important in the beginning when many field values were truly unknown. It made it clear that fields labeled UNKNOWN had not yet been assigned a correct value, as opposed to those fields which were not supposed to have a value (e.g., a host which has no host name alias or where the ethernet address is of no concern will have NONE in the respective fields). The original incarnation of this file had many UNKNOWN's which over time were replaced with their correct values.

Distdata File

The **distdata** file, also a flat text file, contains the *per file* information for those files under *fdist*'s control, one source file per line using the colon (':') as field separators. For each file that is to be distributed, its source name (where it is located on the distribution machine), destination name on the client and distribution domain are all recorded. The same line also contains the distribution frequency, any shell command to be executed on the client after the file has been received and what editor should be used when editing the source file (using EDITOR to indicate the editor specified by either the $EDITOR or $VISUAL environment variables and NONE for no editing allowed).

Config File

The **config** file contains information about the how *fdist* system is configured. **Config** is a text file, with a format inherited from a previous version of *fdist* and whose structure has really outlived its usefulness. **Config** records which host is the distribution master (since commands which alter data are only allowed to execute on the master), as well as a description of the legal domain component names. Under domain component information, we also record which hardware platforms support a given operating system (e.g., ULTRIX does not run on a non-DEC hardware platform).

Distributed Files

Fdist keeps a private copy of all files which are distributed to both guarantee authenticity and provide centralized control of the source files. These source files are kept in an almost flat directory space. Almost, because there is one additional directory level, but this was only implemented because of the legendary inability of UNIX to deal with very large directories. The files were therefore spread out into 60 directories, named 00 to 59. (A better directory naming scheme could have been used but, since the operator is shielded from the path names as all access to the source files is via *fdist* commands, the directory names were kept terse). A new file will be placed into the directories in a pseudo random manner. The destination directory is chosen based on the number of seconds elapsed since the last full minute. This ''poor man's'' approach to random distribution has worked well; currently no directory has more than 12 files in it (with over 400 files being distributed). To further avoid collisions among files with the same name but which are distributed to different domains, a sequence number is appended to the file name when it is installed into the data directory.

Command Interface

The command interface enables the system administrator or operator to create new domains and specify files which are distributed to those domains. It is also possible to modify the content of a text file or to change a file's access permissions, in a manner similar to traditional UNIX commands.

The format of the command line follows what was the current IEEE POSIX 1003.7 draft at the time the project was started. Since the standards group has shown no progress in this area since then, no changes have been made to the original strategy. (Future versions of *fdist* may or may not continue to track the standard.)

With a few exceptions, all commands to the *fdist* system are invoked with a single *fdist* command of the form:

```
fdist -o <command> [<options>] \
      [...] [<argument>] [...]
```

A brief overview of the existing *fdist* commands is located in Appendix A.

Reporting

One problem became very apparent after the first version of the *fdist* system had been installed and running for a few weeks. The output produced by *rdist*(1) was both verbose and generated on a number of machines. There was too much information in a format not fit for human consumption. It was necessary to implement a message processing system which extracted the important messages and presented them to the system administrator in a fashion which matched the domainpaths used by the *fdist* command line interface.

An attempt to write a simple parser failed miserably. The introduction of the domains made it difficult to determine if all hosts had received the files destined for them. The problem was further complicated by the fact that the distribution was spread out over several hosts in order to achieve reasonable performance.

Besides dealing with verbose reporting, it was critical that the distmaster be notified in a timely fashion if something went wrong with the distribution between the master and the distribution slaves.

Efficient reporting had the following requirements:

- Ensure that the master need not rely upon any other machine.
- Ensure that the distmaster be notified by e-mail if the distribution master or slaves encountered problems.
- Analyze the distribution results in the domain context already used by *fdist* and report problems with minimum verbosity.
- Split the error information into daily and hourly reports, where the hourly reports have only the most urgent information.

Notification of Problems

Since it was so important that the distmaster be notified quickly of any problems with either the master or any of the slaves, the mechanism needed to be both simple and robust. Therefore, the mechanism was based on time stamp files. While this approach is not very elegant, it has proven to have the required robustness. The strategy implemented is described below:

- Notify the distmaster if a distribution slave no longer receives updates from the distribution master. This happens if the distribution master crashes, severe networking problems occur, or if the distribution slave is no longer recognized as such by the master.
  - Before doing a distribution to the distribution slaves, the master updates a time stamp file with the current date and time. This file is located within the distribution tree so it is always distributed to each of the slaves.
  - Each time a distribution slave starts a distribution to its clients, it checks whether the time stamp file from the master has been updated within the last 75 minutes. If the file has *not* been updated, the slave notifies the distmaster of the problem by e-mail.
- Notify the distmaster if a distribution slave no longer updates the clients. This happens if

the slave crashes or has networking problems.

- After completing a distribution to the clients, the distribution slave updates a time stamp file with the current date and time.
- Every hour, the distribution master checks each of the distribution slaves, to ensure that no time stamp is older than 75 minutes. If the file has *not* been updated on any of the slaves, the master notifies the distmaster of the problem by e-mail.

The above strategy ensures that the distmaster is always notified when there are problems. The drawback is that when the master goes down, a large amount of e-mail is sent (one message from each slave every hour). However, it was not worth trying to eliminate the duplicate mail in this situation, as it complicated the design, ultimately at a cost of reduced robustness.

Notification of Problems Distributing to Clients

Because the distribution to the clients is, in itself, distributed, we chose to use syslogd to redirect all rdist output back to a common location on the distribution master. All messages are also kept on each distribution slave, in case they are needed for debugging purposes. To simplify later processing, all rdist error messages which span multiple lines are edited on the fly to be only a single line. This editing is necessary as syslogd cannot guarantee that such lines will arrive on the master without being interspersed by other lines.

On the master, each rdist message line is analyzed. One of the goals was to be able to accurately report whether or not a host had been updated with a specific file. Therefore, when a file is updated on the master (e.g., through an edit command), this information is recorded in a dbm database file where the key is the real path and the data is a list of hosts which are supposed to receive the updated file. Each time a message reaches the master that a host has received a specific file, that host is removed from the list of outstanding hosts. This strategy made it possible to implement the **status** command, which allows the system administrator to see which hosts still need to receive a given file.

All error messages are collected and processed. The system administrator can specify whether a given message should show up in either the hourly or daily reports or just simply be discarded. This was implemented using two files, one called filter.delete and the second, filter.day. Each file can contain any number of regular expressions. Error messages matching the regular expressions in filter.delete are discarded; the ones matching the regular expressions in filter.day are placed in the daily report. Messages matching neither set of regular expressions are presented in the hourly report. This approach ensures that new and unexpected error messages are presented to the distmaster with the least possible delay.

Distribution Delays

A distribution scheme which uses rdist has inherently longer propagation delays than some of the more well known distribution mechanisms such as NIS (formerly YP). The various delays introduced by our method were one of the concerns in the overall design. Propagation delay has been minimized as much as possible, especially on those files distributed hourly.

Experience with *fdist* has shown that its longer propagation delay versus NIS has not been much of a problem due to the kinds of data being distributed. However, using rdist as a distribution mechanism does have some significant advantages over the NIS approach:

- Rdist can handle any kind of file, including binary executables.
- The impact of the unavailability of one of the distribution slaves, or even the distribution master itself is far less severe than the unavailability of an NIS server.

Implementation

When we started the design of *fdist*, perl 3.0 had just been recently released. Although neither of us had a lot of experience with perl, we decided to use this as our programming language. It seemed a faster implementation vehicle than C; the shell/awk/sed combination would give us neither the flexibility nor execution speed required.

Looking back, this decision was invaluable to the success of the project. The current implementation of *fdist* is bigger and much more comprehensive than what we had originally planned for. Its implementation relies heavily on the builtin regular expression and string manipulations found in perl. On the negative side however, we found that understanding how programming styles influence the performance of a perl script is, at best, counterintuitive. It would have been very helpful if perl 3.0 had included a profiler, especially since we believe in the methodology which says "make it work, before you make it fast." There were times when working on increasing the performance of the program resulted in runtime changes from hours to minutes or even seconds! Although a lot of work has been done on optimizing performance, there is still ample room for improvement. The two areas which need the most attention are the replacement of the text configuration files with a commercial database and the replacement of the domain name handling routines with ones that use a better approach.

As the *fdist* program grew to its present size of approximately 8000 lines, it became difficult to handle as one large file. As a result, we took an approach similar to C programming where each subroutine (or a set of closely connected subroutines) resides in a separate file. They are then compiled and linked together to form a single program. The difference with our approach is that we replace compilation and linking with concatenation and syntax checking of the individual perl files prior to their execution. (Future versions of fdist may simply use the require() function instead).

The current implementation of *fdist* consists of 92 individual perl files which are concatenated into a number of programs or program segments before they are installed.

A list of the more important subroutines, together with a short explanation of their function is located in Appendix B.

There are also a number of subroutines that read and write the configuration files. As *fdist* evolved, we needed to treat the configuration files as databases with a number of indices. A CPU/memory tradeoff was done where the data is internally replicated for each type of index. This approach does not reduce the initial access time of the files, but it significantly reduces the time for additional accesses. For example, the first time the hostdata file is read, it takes about eight seconds. It is stored in an associative array with a simple counter as the key. Later, if the same data needs to be accessed with, for example, the primary hostname as the key, a new copy of the data will be created as an associative array. This operation takes one to two seconds. Furthermore, access to an associative array which has already been allocated is very fast. While this overall approach is sub-optimal, it is the best one we have been able to implement to date.

Future Work

Much of the future work needed by *fdist* has already been referenced. The configuration files must be replaced with a commercial database in order to achieve better performance and the domain handling routines need a complete rewrite to use a table driven mechanism.

However these items are just a better implementation of what is already in place today. One new item involves using multiple types of file distribution mechanisms. Currently all file distribution is done using rdist. Therefore, *fdist* is limited to "pushing" files from a central host. This method has worked well for system configuration files like */etc/hosts* or */etc/group*, but is not as well suited for other kinds of distribution. There is a need for a "request service," where a host can request a certain application (e.g., elm) and then receive the necessary updates. This method, referred to as polling, would permit users to receive upgrades to their workstations whenever they feel the timing is right. It is important to realize that these two methods must complement each other; both are needed in today's environment.

Also useful would be a subscription type of service where a host can request to receive updates of a file whenever updates are available. This method is similar to the file pushing already in place, but unlike the current method, it allows for making distribution decisions on a host-by-host basis.

It would be advantageous to have a much more flexible domain naming scheme. It is hoped that this flexibility will be realized when the domain name handling routines are rewritten.

Finally, it would be useful to allow distribution of entire directories (e.g., */usr/local*). However, distribution of directories has a number of conceptual problems which, so far, have not yet been resolved.

Shortcomings

In addition to the limitations referenced above, two other points worth noting are:

- A new top level domain cannot be created without making modifications to the *fdist* sources because, at present, the *fdist* system lacks the functionality to make the required changes to the hostdata file.
- The system is dependent on one central distribution server.

Experience With the System

The MIPS user community consists of over 500 users, approximately 400 workstations (mostly dataless) and about 120 servers containing greater than 300 Gb of backed-up, on-line storage across a six building network.

Version 2 of *fdist* has been in use at MIPS for over a year and has grown to where almost all hosts and over 400 files are maintained under the system. Files are easily added, removed or updated, and placed under *fdist*'s control.

The reports enable the system administrator to identify both quickly and easily any problems relating to the distribution, ranging from the simple cases of a client being down or out of disk space to the more severe cases of problems with one of the distribution slaves or even the distribution master itself.

As a result of using *fdist*, MIPS has been able to provide a distributed, homogeneous environment to its user community; one where the user no longer needs certain machines to accomplish specific tasks. In fact, the environment has become so homogeneous in appearance, that many users have become unaware of which machines they specifically rely on beyond the workstation on their desk.

Availability

The current version of *fdist* is 2.1. *Fdist*'s availability is presently handled on a case by case basis. Contact Paul M. Moriarty at MIPS for additional information.

Acknowledgments

- We want to thank Larry Wall for the creation of perl. Without perl, this project would probably never have been attempted.
- Thanks to Tom Christiansen for helpful suggestions on how to improve the performance of the messages filter, to a point where the tool became usable.
- Finally, thanks to Susan Woolf and Rob Kolstad for proofreading this paper.

Author Information

Paul M. Moriarty, a senior systems administrator in Engineering Computer Services, has been with MIPS since 1988. He is responsible for system administration automation and utilities as well as postmaster and news administrator for the whole company. Paul is co-founder of Bay-LISA, a San Francisco Bay Area user's group for system administrators of large sites. Contact Paul via US Mail at MIPS Computer Systems, Inc; 928 Arques Avenue, M/S 1-06; Sunnyvale, CA 94086 or via e-mail at pmm@mips.com or {ames, decwrl}!mips!pmm.

Bjorn Satdeva is the President of /sys/admin, inc., a consulting firm which specializes in Large Installation System Administration. Bjorn is a member of the IEEE POSIX 1003.7 System Administration Standardization Committee. Bjorn is also President and co-founder of Bay-LISA, a San Francisco Bay Area user's group for system administrators of large sites, and Senior Editor for ROOT, the UNIX System Administration Magazine. Contact Bjorn via US mail at /sys/admin, inc.; 2787 Moorpark Avenue; San Jose, CA 95128 or via e-mail at bjorn@sysadmin.com or the UUCP address {uunet, mips}!sysadmin!bjorn.

References

- IEEE POSIX 1003.7 System Administration, Draft 2.
- RFC1034 Domain Concepts and Facilities

Appendix A

Fdist Commands

What follows is a brief overview of the existing *fdist* commands.

Maintaining Domain Information

- Create a new domain

  Synopsis: *fdist -o create <new-domain>*

  The **create** command creates a new, empty domain. The parent domain must already exist. The command enters the necessary lines in the config file to allow files to be added to the new domain as necessary. If a domain with a similar configuration already exists, it is probably simpler to clone that domain rather than create the new domain from scratch.

  This command can only be executed by the super user.

  To create a new domain *new* under the domain osgrp.adm:

  ```
  fdist -o create new.osgrp.adm
  ```

- Delete a domain

  Synopsis: *fdist -o delete [-f] <domain>*

  The **delete** command deletes an existing domain. The domain is required to be empty unless the -f option is specified, in which case all files and sub-domains contained by this domain are also deleted.

  This command can only be executed by the super user.

  The following command removes the domain obsolete.osgrp.adm:

  ```
  fdist -o delete obsolete.osgrp.adm
  ```

- Clone an existing domain

  Synopsis: *fdist -o clone <domain> <new-domain>*

  The **clone** command ''clones'' an existing domain into a new domain. The new and the existing domain must be on the same level. The clone command creates the new domain in the config file and creates new entries for each file in the new domain against what is found in the old domain. No changes are made to the old domain. The following command creates a new domain new.osgrp.adm, by cloning the domain old.osgrp.adm

  ```
  fdist -o clone old.osgrp.adm new.osgrp.adm
  ```

- Compare two domains and note the differences

  Synopsis: *fdist -o compare <domain1> <domain2>*

  The **compare** command compares two domains and lists the differences. It reports the results in five groups:
  - Files distributed to <domain1>, but not to <domain2>;
  - Files distributed to <domain2>, but not to <domain1>;
  - Files distributed to both <domain1> and <domain2> which are different;
  - Files distributed to both <domain1> and <domain2> which are identical but have different source files;
  - Files distributed to both <domain1> and <domain2> which are shared (same source file).

- Print a list of domains to which a host belongs

  Synopsis: *fdist -o domain [-F] <host> [...]*

  The **domain** command prints a lists of all domains to which the host is a member. The -F option specifies the additional reporting a list of all files that the host will receive (with distribution frequency shown).

- Print list of hosts in a domain

  Synopsis: *fdist -o hosts [-F] <domain> [...]*

  hosts prints a lists of all hosts in a domain. If the -F option is specified, it also writes a list of all files that the host will receive (with distribution frequency shown).

- Print a list of files distributed to a domain

  Synopsis: *fdist -o list <domain> [...]*

  The **list** command prints a list of all files distributed to a domain. The following command below shows what files are distributed to the domain cc.adm

```
$ fdist -o list cc.adm

Files distributed to domain osgrp.adm:
day    osgrp.adm:/.forward
hour   common.adm:/.rhosts
week    common.adm:/etc/TIMEZONE
hour   common.adm:/etc/hosts
hour   common.adm:/etc/hosts.equiv
```

Maintaining File Information

- Install a new file into a domain

  Synopsis: *fdist -o install [-e <editor>] [-s shell_command]*
  <file> <domainpath> <frequency>

  The **install** command adds a new file to the distribution tree. If the new file is already distributed to another domain, its domainpath can be used in place of the file argument. However, it must be noted that this creates a separate, but identical file. If the two files must be shared, they must be merged using the **mrg** command (see below).

The -e option sets the editor to the specified value. The default is to use the values (at the time of editing) of the $VISUAL or $EDITOR shell environment variables. If neither of these are set, vi is used. If the file is not a text file, the default value is no editing.

The -s option sets the shell command to the specified value. This shell command is executed on the client after the file has been distributed. The default is no shell command.

- Print a list of domainpaths to which a file is distributed

Synopsis: *fdist -o file [-a] <file1> [<file2> ... <filen>]*

The **file** command prints a lists of all domainpaths to which a file is distributed. This is useful for files shared by multiple domainpaths as one can see exactly which domainpaths share the file. The -a option will display the distribution frequency and optional shell command together with the domainpaths. The command below shows what domains share the file /usr/lib/aliases:

```
$ fdist -o file /usr/lib/aliases

File /usr/lib/aliases is distributed to:
disked.sun3.sunos.common.adm:/usr/lib/aliases
400.common.adm:/usr/lib/aliases
bsd.common.adm:/usr/lib/aliases
450.common.adm:/usr/lib/aliases
ultrix.common.adm:/etc/aliases
```

- Set the distribution frequency for a file

Synopsis: *fdist -o freq <domainpath> <frequency>*

The **freq** command changes the distribution frequency of a file.

- Merge two domainpaths into one

Synopsis: *fdist -o mrg <domainpath1> <domainpath2>*

The **mrg** command merges two domainpaths together, changing the real file in domainpath2 to be the same as domainpath1. All other entities for domainpath2 remain unchanged (e.g., if domainpath2 has a shell command

assigned, while domainpath1 does not, domainpath2 will still have the shell command after the merge command has been executed).

The following command results in domain1.adm:/somefile being the same (shared) as domain2.adm:/somefile.

fdist -o mrg domain1.adm:/somefile domain2.adm:/somefile

- Select what editor should be used

Synopsis: *fdist -o setedit <domainpath> [<editor>]*

The **setedit** command changes the editor used to <editor> whenever the <domainpath> file is edited using the fdist -o edit command. If the editor field is omitted, the editor path is set to the default.

- Specify shell command to execute after file distribution.

Synopsis: *fdist -o shell <domainpath> [<shell>]*

The **shell** command modifies the shell command for a file. This command is executed on all hosts that <domainpath> is distributed to, every time that file is updated. If no shell command is specified, any existing shell command will be removed.

- Edit a file

Synopsis: *fdist -o edit [-e <editor>] <domainpath>*

The **edit** command invokes an application specific editor on the file specified by the domainpath given as its argument. The editor used depends on which editor was specified when the file was installed (or later changed by fdist -o setedit). In each case, the domainpath is be replaced by the real path before editing.

The actual editor used depends on a number of factors.

- If the Editor field in the *fdist* database starts with '/', it is taken to be a full pathname of a program which will be executed.
- If the Editor field is EDITOR, a text editor is invoked. If the environment variables $EDITOR or $VISUAL are set, then this value is used to select the editor. Otherwise the default is the vi editor.
- If the Editor field is NONE, no editing is allowed.
- If an editor is specified using the -e, that editor is used instead of the editor specified in the **distdata** file. This feature is intended to be used in scripts to override an interactive editor with one that is non-interactive.
- All other field contents are taken as a command name to be invoked using the normal UNIX search path.

The file to be edited is locked before the editor is called using a simple file lock that is honored by all *fdist* commands. If no changes are made to the edited file then the date of last modification remains unchanged. This prevents unnecessary redistribution of unmodified files.

- Unlock a locked file

Synopsis: *fdist -o unlock <domainpath>*

The **unlock** command unlocks a file that has been unintentionally left locked (e.g., after a system crash) by removing the lock file.

Traditional File Manipulation

All the commands in this group are, in fact, standard UNIX commands with the necessary packaging around them to make them work correctly in the *fdist* environment. The processing which is done checks for valid parameters since not all parameters are permitted in the *fdist* environment (e.g., since *fdist* does not yet support the notion of directories, the -r option on the rm command is not meaningful).

- Compare two files

  Synopsis: *fdist -o cmp <domainpath1> <domainpath2>*

  The **cmp** command compares two domainpaths by first converting the domainpaths to the real paths. After conversion, cmp calls the UNIX **diff** command for text files; the UNIX cmp command for all other files. If only one of the two domainpaths is a text file, the following messages are displayed:

  cmp: <domainpath1> is a text file
  cmp: <domainpath2> is a binary file

- Concatenate and print files

  The **cat** command converts all domainpaths to the real paths, and then calls the UNIX cat command, passing any command line arguments to the UNIX command.

- Change group,
  Change mode,
  Change owner,
  Copy a domainpath,
  List information about a file,
  Remove a domainpath

  The **chgrp**, **chmod**, **chown**, **cp**, **rm** and **ls** commands convert all domainpaths to the real paths and then call their respective UNIX command, passing any command line arguments to the UNIX command.

Maintaining rdist Information and Distribution

While all the commands mentioned above are aimed at maintaining domains and files, it is also necessary to initiate the distribution by calling rdist. There are three commands for this:

- Build distribution control files

  Synopsis: *fdist -o build [-d <domain>] <frequency> [-f <file>]*

  The **build** command builds distribution control files but does not initiate the distribution of any files. The <frequency> parameter specifies for what distribution frequency the files are rebuilt. If no options are specified, the output is written to a file where the dist command later finds it. Note that both the f and d options result in redirection of that output. The output is written in a format that is a complete command file for the *rdist*(1) utility.

  If the -d <domain> option is specified, the build command only generates distribution information for files distributed to that domain. The resulting information is written to standard output, unless the -f option is specified.

  If the -f option is specified, the build command redirects the distribution information to the specified file.

  The following command line rebuilds the distribution control file for the hourly distribution. The output is placed where 'fdist -o dist hour' later finds it.

  ```
  fdist -o build hour
  ```

  The next script writes a distribution control file to stdout, which contains entries for all files distributed daily to the domain sysv.cc.adm

  ```
  fdist -o build -d sysv.cc.adm day
  ```

- Distribute files

  Synopsis: *fdist -o dist [-v] <frequency> [<host>] [...]*

  The **dist** command initiates the distribution of the files distributed with the interval specified by <frequency>. This is done most conveniently by executing fdist -o dist at the correct time intervals from cron. Fdist -o dist initiates the fdist -o ldist command on all distribution slaves.

  If the frequency is specified as *all*, a distribution for all frequencies is done.

  One or more hosts may be optionally specified, in which case only the specified hosts receive file distribution. The use of *all* conjunction with a hostname is convenient for updating new hosts that are brought online.

- Distribute files from local distribution slave

  Synopsis: *fdist -o ldist <frequency> [-r] [<host>] [...]*

  The **ldist** command initiates the distribution of the files distributed with the interval specified by <frequency>. Ldist is most often executed on a distribution slave as a result of an fdist -o dist command having been executed on the distribution master.

  Since no data is locally modified on the distribution slaves, this command does not build distribution control files.

Ldist, like dist, accepts *all* as a frequency and can limit its distribution to single hosts.

The -r option causes the ldist command to execute silently as a background process.

Miscellaneous Other Commands

- Check for last distribution

  Synopsis: *fdist -o check [-cmqv]*

  The **check** command checks a distribution slave for its distribution status and writes the results to stdout. The -c option causes check to ignore client distribution status. The -m option causes check to notify the distmaster by e-mail if the distribution slave has not been updated for the last 75 minutes. The -q option suppresses any messages that check would print. The -v option causes check to always print the distribution status independent of the time period since last distribution.

  Since check is used to notify the distmaster of missing file distribution from the distribution master, the following command should be executed hourly from cron by root:

  ```
  fdist -o check -cmq
  ```

- Display the hostdata file with new slave information

  Synopsis: *fdist -o chslave <from> <newslave>*

  The **chslave** command substitutes one distribution slave name in hostdata with another and writes the results to stdout. The hostdata file is left unmodified. The <from> argument can be a subnet IP address (e.g., 130.62.6) instead of being a distribution slave name, in which case all hosts on that subnet become clients of newslave.

- Display hostdata in ethers format

  Synopsis: *fdist -o mkether*

  The **mkether** command writes the information in the hostdata file to stdout in the same format as that used in the */etc/ethers* file. Duplicate ethernet addresses and hostnames are reported on stderr. Mkether makes no attempt to correct such errors. Therefore, if any duplicates exist in the hostdata file, they will also appear in the output of this command.

- Display hostdata in hosts format

  Synopsis: *fdist -o mkhost [-s]*

  The **mkhost** command writes the information in the hostdata file to stdout in the same format as that used in the */etc/hosts* file. Duplicate IP addresses, hostnames and host aliases are reported on stderr. Mkhost makes no attempt to correct such errors. Therefore, if any duplicates exist in the hostdata file, they will also appear in the output of this command.

  Because some network components use the same hostname for multiple IP addresses, the reporting of duplicate hostnames can be turned off by specifying the -s option. Duplicate IP addresses and host aliases are still reported.

- Display slave-client relationship

  Synopsis: *fdist -o network [<slave>] [...]*

  The **network** command prints a list of clients for each distribution slave, sorted by network address. By default, it prints information for all distribution slaves. If one or more distribution slave names are provided as arguments, only information for those specified distribution slaves is printed.

- Generate distribution report

  Synopsis: *fdist -o report hour|day [<time>]*

  The **report** command prints a distribution report. If the first argument is hour, it prints an hourly distribution report; if the first argument is day, it prints a daily distribution report.

  By default, the command prints the report for the current day/hour, unless the optional <time> argument is specified. <time> is an integer, which specifies the time period of the report to be printed (that day in the month or that hour in a 24 hour period). For example,

  ```
  fdist -o report hour 8
  ```

prints the 8 AM distribution report. If the time period specified is larger than the current time period, the report from the prior day/month is printed.

<time> can also be specified as a negative integer, in which case the command prints a report from a previous period relative to the current period (e.g., fdist -o report day -1 prints yesterday's daily distribution report).

Note the daily and hourly reports have no connection to the distribution frequencies used for individual domain paths. The two reports contain messages from *all* distribution frequencies which are split into the two report categories, depending on their estimated importance. See the section on reporting below.

- Display distribution status

  Synopsis: *fdist -o status [<domainpath>] [...]*

  For each domainpath specified, **status** prints a list of all hosts which still have yet to receive the latest version of a specific source file. By default, the status command prints the distribution status for all domainpaths and all hosts which have not received the latest copy of the given domainpath.

  If one or more domainpaths are given as argument, only the status for those domainpaths is displayed.

  The data is based on the information kept in the dbm file called the *missing database*.

Appendix B

Fdist Subroutines

Below is a list of the more important subroutines, together with a short explanation of their function. The domain name manipulation routines all support the use of wildcards, but this functionality can be turned on or off using a boolean parameter.

- *DomainCompare* compares two domain names and determines if they are identical (allowing for use of wildcards in the domain names). For example sysv.common.adm is equal to sysv.osgrp.adm (since "common" is our wildcard domain name), but bsd.common.adm is not equal to sysv.osgrp.adm
- *DomainContained* compares two domain names to verify if the first domain names is contained in the second (allowing for use of wildcards in the domain names). For example, sysv.osgrp.adm is part of osgrp.adm, but osgrp.adm is not part of sysv.osgrp.adm
- *DomainCreate* creates a new domain, performing all necessary tests to ensure the new domain is correct.
- *DomainDelete* deletes an existing domain.
- *GetDomainPath* maps a real path for a specific host back into a domainpath.
- *DomainVerify* checks a domain name to ensure it is valid and exists (allowing for use of wildcards in the domain name).
- *FullDomains* expands any wildcards in a domain name. The result is a list of domain names which match the wildcards. *Open* and *Close* are routines which are used to open and close files when file locking is required. A simple locking mechanism is used to create a lock file. This eases integration with shell scripts when required. With *fdist*, the following lock requests are supported:
  - Read-only lock: Open the file for read, without creating a lock file.
  - Open for Write after Read: Establish a lock on the file and then open the file for read. Fails if the file lock cannot be established.
  - Open for Write: Used to open the file for writing when the lock file has already been established by an Open for Write after Read.
- *GetRealDomain* returns the official domain path as it is used in the database (including wildcards). For example, when called with sysv.osgrp.adm:/somefile as argument, it may return sysv.common.adm:/somefile
- *GetRealPath* returns the full source pathname corresponding to a domain path. For example, sysv.osgrp.adm:/etc/group may return /fdist/data/37/group.2

Appendix C

Sample Fdist Command Written in perl

This example shows the delete command, which deletes a domain.

```
#  main.pl contains the main subroutine, delete, for the delete command
#  This is called from fdist
#
sub delete {

        #
        # Standard scheme for argument parsing is
        # All options, Argument list

        local( $Options, @ArgList ) = @_ ;

        local( $OldDomain );
        local( $Force );

        @ARGV = @ArgList;


        #
        # Set the default options
        #
        $opt_F = undef;


        #
        # And call Getopts from the perl library
        #
        do "getopts.pl" ;
        if ( ! do Getopts( $Options ) ) {
                do UnlockConfig();
                return( $False );
        }

        if ( $#ARGV != 0 ) {
                # We must have exactly one argument
                do CmdUsage( "delete" );
        }

        #
        # Initiating the variables from command line
        #

        $OldDomain =    @ARGV[ 0 ];
        $Force =        $opt_F;

        #
        # Verify the domain exists
        #
        if ( ! do DomainVerify( $OldDomain, $False ) ) {
                if ( $Parrent =~ /common/ ) {
                        do PrtError( $NoWild );
                }
                else {
```

```
                do PrtError( $NoDomain, $OldDomain);
        }
        do UnlockConfig ();
        return( $False );
}



#
# Read the host file, and ensure there are no hosts in the
# domain to be deleted
#
printf( "Reading the host database.  Please wait ...\n" );
%HostList = do ReadHostData( 'domain' );
printf( "Thank you\n" );


foreach $Key ( keys %HostList ) {
        if ( do DomainContained( $Key, $OldDomain, $False ) ) {
                do PrtError ( $NotEmptyDmn, $OldDomain, 'hostdata' );
                do UnlockConfig ();
                return( $False );
        }
}

#
# Read the dist file, and ensure there are no files in the
# domain to be deleted, unless overwritten by the $Force option
#
if ( ! $Force ) {
        %DistList = do ReadDistData( 'domain', 'R' );


        foreach $Key ( keys %DistList ) {
                if ( do DomainContained( $Key, $OldDomain, $false ) ) {
                        do PrtError($NotEmptyDmn,$OldDomain,'distdata');
                        do UnlockConfig ();
                        return( $False );
                }
        }

        if ( ! do DomainDelete( $OldDomain ) ) {
                do UnlockConfig();
                printf( STDERR "Cannot Delete Domain\n" );
                return( $False );
        }
        if( ! do WriteConfig() ) {
                printf( STDERR "Write Config Fail\n" );
                return( $False );
        }
        return( $True );
}
else {
        %DistSrc =  do ReadDistData( 'source', 'R' );
        %DistData = do ReadDistData( 'domain', 'RW' );

        #
        # Sequence is important here
        # We cannot undo the removal of files if the
        # the deletion of a domain fails. We will
```

```
                # delete the domain first, and then the files.
                #

                #
                # Remove $OldDomain from Config
                #
                if ( ! do DomainDelete( $OldDomain ) ) {
                        do UnlockConfig();
                        do UnlockDistData();
                        printf( STDERR "Domain Delete Failed\n" );
                        return( $False );
                }

                #
                # Remove $OldDomain from DistData
                #
                foreach $Key ( keys %DistData ) {
                        if ( ! do DomainCompare( $Key, $OldDomain, $False ) ) {
                                next;
                        }
                        delete $DistData{ $Key } ;
                        foreach $Value( split( ' ', $DistData{$Key} )){
                                ($Destination, $TmpDomain, $Source ) =
                                        split( ':', $Value );

                                @Source = split( ' ', $DistSrc{$Source}  );

                                if ( $#Source == 0 ) {
                                        $Path = sprintf( "%s/%s",
                                                $DataDir, $Source );
                                        system( "rm $Path"  );
                                }
                        }
                }

                #
                # We are updating two data files.  Make sure to back out
                # correctly if we fail to write to them both.
                #
                if ( do WrtDistData( %DistData ) ) {
                        if ( ! do WriteConfig() ) {
                                rename( ${DistData}, ${DistData}.".bad" );
                                rename( ${DistData}.".old", ${DistData} );
                                system( "rm", $DistData."bad" );
                                do UnlockConfig();
                                return( $False );
                        }
                }
                else {
                        do UnlockConfig();
                        return( $False );
                }
                return( $True );

        }
}
```

Link Globally, Act Locally: A Centrally Maintained Database of Symlinks

Arch Mott - MIPS Computer Systems, Inc.

ABSTRACT

Large computer installations using "file server" and "compute server" paradigms face a challenging dilemma: Do systems administrators force already overburdened computer users to remember twisty pathnames to find the way to their NFS accessed data, or resign themselves to a frustrating life of hand-maintaining a burgeoning array of symlinks on each machine under their control to provide the user with an easily remembered path to the data. Common sense dictates that there must be a middle ground which provides an easily maintained, user-friendly solution to both sides of the problem.

Introduction

Maintaining a homogeneous working environment across a heterogeneous workspace involving many servers and workstations is a difficult task. This became evident maintaining the large numbers of symbolic links that provided hardware and software engineers with a uniform development environment across many machines of differing architectures and operating systems. This particular problem was solved by developing a centrally maintainable database of symbolic links which is distributed to the servers and implemented (or reimplemented) there on a regular basis.

The "Symlink" system provides the system administrator with a powerful and flexible tool for the maintenance of the above mentioned environment. The database file itself takes the form of a flat ASCII text file and the executable, which builds the actual symbolic links, is a Bourne Shell script. The script is written to allow the specification of alternate database files, and in the special case of users' home directory links, some rudimentary verification of the validity of that home directory with respect to the password file.

Design Goals

The "Symlink" system was originally designed in a non-NIS environment. It provided a facility for the maintenance of a single directory structure containing links to all users' home directories. This directory, insightfully named "/user", needed to be present on all machines on the network. This setup allowed a user to log in to a machine anywhere on the net and easily find their home directory (all home directories in the password file take the form "/user/<username>") as well as the home directories of other users.

The Database

The database developed to fit the above criteria is a flat ASCII text file containing the target of the symbolic link and the name of the symbolic link, one link entry per line with the data fields separated by white space. Optionally, a hostname or OS type may also be specified to provide the ability for special exception handling based on that information. Since the system was designed with home directories in mind, it assumes that a non-absolute pathname creates a link built in the default "user" directory; Any entry with a leading "/" results in a link built in that absolute location.

The typical case of a link to be created globally would look like this:

```
/hwdesign /n/talos/development/hwdesign
```

This link is built on all machines controlled by the "Symlink" system and will point from /hwdesign to the nfs mounted directory /development/hwdesign from the host "talos".

A brief example of what a database entry for the symbolic link "/eng" might look like is shown below.

```
/eng /n/barsoom/devl/eng UMIPS-BSD
/eng /n/trantor/eng-dir/eng trantor
/eng /n/tazenda/data5/eng
```

In this example, the "tazenda" link is the default, the "trantor" link is specific only to the host trantor, and the "barsoom" link is built on any machine running the UMIPS-BSD operating system. In any case, the user can reference files in the "/eng" directory and arrive at a predetermined location without knowing the real absolute pathnames.

Here is an example of what a user's home directory entry would look like:

```
fred  /homedirs/fred barsoom
fred  /n/aurora/homedirs/fred
```

In this example, a link called "fred" is built in the default "user" directory on the machine barsoom

which actually references a directory local to the machine (/homedirs/fred) , where all other hosts would have in their "user" directory a pointer to /n/aurora/homedirs/fred. This particular usage is very convenient if a user does work for a particular project on one machine which requires a unique environment, but wants their standard environment everywhere else.

(In all above examples, "/n" is used as a standard directory on all machines or host-by-host NFS mount directories.)

Distribution of the database is handled using "fdist", an rdist based system.

The Script

The script is also distributed using the "fdist" system to install the script where it does not already exist (e.g., newly installed systems) Execution of the script is handled via cron, or any other chosen periodic job scheduler.

Wins

In environments where NIS is not available or is considered politically incorrect, the "Symlink" system provides an easily maintainable method for accessing NFS mounted home directories network-wide. In fact, the original concept predates NIS and was conceived to provide this service. Its functionality, however, is not limited to home directory access.

Losses

True System V systems do not have access to the symlink system call, which invalidates the entire concept.

Availability

The "Symlink" system is available on a case by case basis from the author.

Author Information

Arch Mott is a Senior Systems Administrator at MIPS Computer Systems and is on the Board of Directors of Bay-LISA, the San Francisco Bay Area LISA group. He may be reached at MIPS Computer Systems, Inc. 928 Arques Avenue, Sunnyvale, CA, 94086, Mail Stop 1-06, (408) 524-8256. Email amott@mips.com

Watson Share Scheduler

Carla Moruzzi & Greg Rose - IBM T.J. Watson Research Center

ABSTRACT

A Compute Power Server Cluster consists of a number of loosely coupled RS/6000 work stations using shared file systems and CPU resources. Resource management is necessary to ensure that users receive access to resources without interference from other users. The Watson Share Scheduler allows each user a fair share of the resources of the cluster. The scheduler ensures that the resources are distributed fairly across an environment consisting of both numerically intensive and interactive users. Interactive users see reasonable response times while CPU bound processes are consuming cycles. The scheduler also ensures that each process receives some resources so that no process is blocked. The Watson Share Scheduler provides an environment which enhances system performance and simultaneously allows the delivery of reasonable response times to individual users.

Introduction

In addition to a number of compute servers, the Compute Power Server Cluster has associated with it a number of RS/6000 machines which function as file servers, and a cluster master machine. The master machine performs various monitoring and updating functions for the cluster as a whole, and drives the Watson Share Scheduler.

Under AIX, process scheduling causes processes to compete on a per-process basis, making it possible for one user to receive a disproportionately large share of the CPU resources by running a large number of processes. The solution to the problem is to make the CPU share on a per-user basis rather than per process.

A number of related articles recognize the need for a share scheduler. Scheduling for a Fair Share of the Machine[1], the Sydney University Fair Share Scheduler[2] and the Comparison of Resource Managers for VM/XA[3] are merely a few.

Design

The Watson Share Scheduler is designed to manipulate process priorities to try to deliver a known proportion of the CPU resources to the active users. However, the known proportions are not necessarily equal; there is a single file (*shares*) used to define these proportions for the special cases.

A number of constraints dictated the design of the Watson Share Scheduler. The Compute Power Server Cluster, a production computation facility for scientists at the IBM T.J. Watson Research Center, runs on the standard AIX operating system for the RS/6000. The standard AIX kernel is not modified in any way, and the scheduler runs entirely outside the kernel using data already available in local accounting facilities. These facilities include process accounting records and the output of the modified version of the *ps* command.

Since the cluster was evolving rapidly, simple networking services were used for the initial implementation. *Rsh* is used to run processes on compute servers to produce files, and the files are then examined on other machines using NFS, ftp, etc.

The user environment which the Compute Power Server Cluster serves consists of a large number of scientists with numerically intensive computation. Interactive use of the cluster is typically limited to program development debugging.

These constraints led to an implementation utilizing small scripts and a modular design. Another description of the resulting Watson Share Scheduler might involve the word "hack". The software is malleable in the face of changing demands, and does not rely on any AIX or RS/6000 specific feature.

Set Up

All cluster users are treated equally and are allocated a number of shares in the same manner as in the Sydney University Fair Share Scheduler. Shares are only effective when a user is active, equal numbers are equal proportions, but the actual percentage is not defined unless the total number of shares outstanding is known.

In the absence of any special consideration, a user is allocated the default number of shares. This default is established in the *shares* file. (There is even a default default of 10 shares.)

Often there are special considerations. For example, a particular user may have a requirement for a fixed share (usually large) of the CPU

[1] "Scheduling for a Fair Share of the Machine", Larmouth, Softw. Pract. Exper., January 1975

[2] "A Fair Share Scheduler", Kay and Lauder, Communications of the ACM, 1988

[3] "A Comparison of Resource Managers for VM/XA", Doherty & Pope, CMG, 1991

resources, for production runs of specialized software. This is handled in the *shares* file by suffixing a percent sign to the number. There is a check to ensure that more than the available resources cannot be used for fixed allocations.

Some users execute programs on behalf of a number of other users, and hence require a larger, but nevertheless varying, allocation. There are also users who work on group projects, and for the purposes of scheduling are considered equivalent.

Provision for these three special cases is available within the Watson Share Scheduler, and can be seen in the example file shown below. Empty lines are ignored, and characters after '#' are comments. Except for this, lines must consist of a user name (or "default") followed by a number of shares or fixed allocation percentage. Following such a line there may be other lines with only a user name, indented, to indicate that this user is equivalent to the previous mentioned user.

Here is the current *shares* file:

```
# the next line establishes a
# default of 10 shares
default  10

# jim needs lots of resources
jim      40%

# calvin works on three projects
calvin   30

# dan is really sam in disguise.
sam      10
   dan
```

Implementation

Most of the Watson Share Scheduler is implemented in perl[4] scripts. There is an enclosing shell script called *do_share* which sequences the three phases described below. Another script called *share*[5] loops to run the *do_share* script every five minutes.

The three main phases of running the Watson Share Scheduler are :

1. Gathering CPU usage data from the compute servers.
2. Collating the usage data and calculating relative priorities.
3. Modifying the priorities of running processes on the compute servers.

Of these, the second phase runs entirely on the cluster master, while the other phases use the *rsh* command to invoke processing on the individual compute servers.

The Watson Share Scheduler assigns new priorities to processes in an attempt to re-apportion the CPU usages. These adjustments are empirical and may over or under compensate; thus forming a feedback loop, when subsequent priority adjustments are made.

1. Gathering CPU Usage Data

Gathering the CPU usage data is accomplished by invoking a local perl script called *getusagesummary* on each compute server in turn. This script relies on the process accounting mechanisms in use on the Compute Power Server Cluster to provide summarized raw data. The output from this script is a very simple text file. Each line of the file represents a different user, and consists of three tab-separated fields: the numeric user id, the total CPU time times the clock speed for the period in mega-instructions, and the number of eligible active processes.

A process accumulates more than 10 seconds of CPU time before it is eligible for priority adjustment. All processes are counted in the total usage, but shorter processes are ineligible for priority adjustment and do not directly affect the priority calculations.

The Compute Power Server Cluster consists entirely of RS/6000 computers, however since they are not all the same model, they run at different speeds. Since they all have the same instruction set, compilers, etc., the same number of instructions are executed to achieve the same computation. The CPU times which are recorded in seconds are therefore adjusted by the rated clock speed of the model in question (as reported by the local command supplied with AIX). While it is debatable just how this relates to other computers or to real work, it is commensurate on all of the RS/6000 models. The result of this adjustment is theoretically the number of millions of instructions executed by the process, so the unit is called mega-instructions (MI).

Because of the way the accounting data is gathered, the period for which the usage is reported is always from midnight some number of days ago, to the current moment. The number of prior days can be adjusted to more than the one day currently used by the Watson Share Scheduler. The period is derived by taking the AIX process accounting records which are written when processes terminate, and supplementing them by records of the same format generated by a modified version of the *ps* command from running processes. A snapshot of running processes is automatically taken at each midnight. The total usage for the period is thus given by $U = R_{now} - R_n + \sum_{i=0}^{n} F_i$ where R_n corresponds to the accounting records generated at midnight n days ago

[4] *Programming perl*, Larry Wall and Randal L. Schwartz, O'Rielly & Associates Inc., 1991.

[5] *Cron*, the UNIX timing mechanism, will implement this feature in the future.

for running processes, R_{now} corresponds to the same kind of records generated now, and F_i is the file of accounting records accumulated i days ago. F_0 is the active file representing data currently accumulating. There is a possibility of disagreement between R_{now} and F_0, because a process may terminate during the short window between accumulation. In practice this is extremely rare and has negligible effect on the scheduling over the periods used because of the feedback nature of the Watson Share Scheduler.

There is an interesting side effect to the way the usage information is gathered. If a compute server crashes, termination records for the running processes are not written and the above formula will discount the wasted instructions. Although this is arguably correct behavior, it can lead to apparently negative CPU usage when a process which has been running for some days is lost through a machine crash. Fortunately, while such processes are common on the cluster, crashes are quite rare.

2. Collating Usage Data And Calculating Priorities

Once the summary usage data has been collected from each of the compute servers, the files are processed by a script called *calcshares*. This script reads the *shares* file described previously and the usage summaries, and performs the following calculations.

1. A default share entitlement is given to users with active processes who are not explicitly mentioned in the *shares* file.
2. Users who are not active, but who are allocated a fixed share of CPU resources, are removed from the calculations. If this is not done, the algorithm would continuously adjust other users' priorities down at a time when they had no competition.
3. Shares are now re-expressed as percentages of available resources, totalling 100%. Similarly, relative percentages of usage, also totalling 100%, are calculated from usages of the active users.
4. The next step is to calculate a "fairness" of each of the users. Since the intention is to use the result of this calculation to influence the *nice* value of the processes, it was desirable that it produce manageable numbers. Thus, the formula log(*%alloc*/*%usage*) was chosen, somewhat arbitrarily.
5. Lastly, new nice values are chosen for the active users. The range of "fairnesses" found is mapped into the interval 1 to *nice_most* (*nice_most* is tunable, currently set to 12). The number of processes being scheduled for a user, minus one, is added to the range. This is in an attempt to compensate for the fact that AIX (and UNIX generally) timeshares between processes, not users, and the more processes a user has running, the more CPU will be allocated.

```
Wed Jun 12 09:30:07 EDT 1991
          Process usage summary for active users
User      %Alloc   Usage(TI) %Usage Cycles Fairness  #Procs Newnice
----      ------   --------- ------ ------ --------  ------ -------
Fred        7.50      0.0012   0.02   0.62        6       1       1
Elmo       22.50         1.4  21.08  28.77    0.065       2      12
Jim        40.00         2.6  39.41  51.38    0.015       5      15
Carol       7.50        0.49   7.48  10.74   0.0029       1      11
Toonce      7.50        0.55   8.29 136.18     -0.1       8      18
Dinsdale    7.50        0.62   9.37   6.30    -0.22       2      12
Hobbs       7.50        0.91  13.75  16.98    -0.61       1      12
```

Figure 1: Display From The Watson Share Schedule Report

User	Total	node1	node2	node3	node4	node5	node6	node7	node8	node9
Toonce	136	13	8	8	12		21	28	28	13
Jim	51	14	9	9	13	4				
Elmo	28			10		18				
Hobbs	16					1				15
Carol	10		10							
Dinsdale	6						6			
Fred	0									
TOTAL	247	27	27	27	25	23	27	28	28	28

Figure 2: Usage Breakdown

After performing these calculations, *calcshares* writes a report detailing the usages and new priorities, so that the users can get feedback about load on the cluster. It also writes a file called *renice* (not to be confused with the AIX command of the same name) which contains lines with numeric user ids and new nice values to be applied to those users.

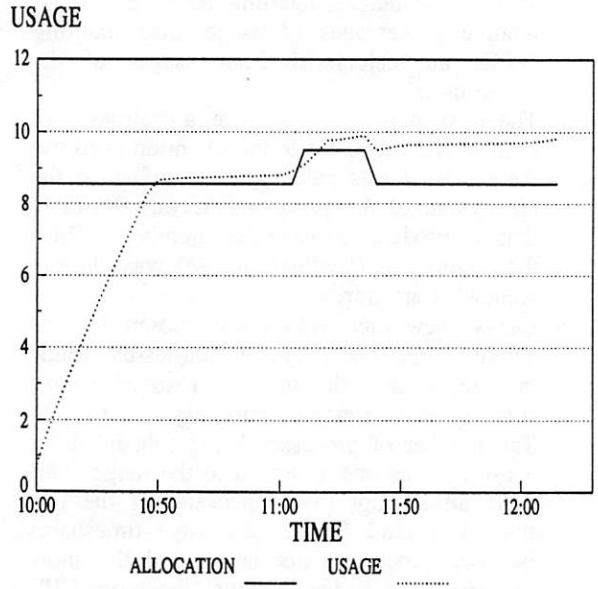


Figure 3: Data sample taken from the Watson Share Schedule Reports

3. Modifying Priorities

A script is run on each of the compute servers on the cluster, with its input being the *renice* file produced in the previous phase. It examines the running processes and executes the *renice* command to assign the desired nice value to them.

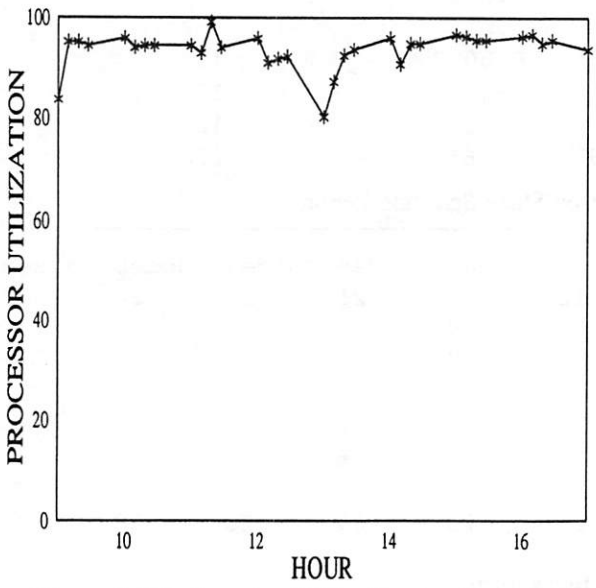


Figure 4: Displays the CPU utilization by users

The previous phase calculates new nice values simply by a formula, without observing limitations of the *nice* mechanism itself. For example, on AIX, *nice* values are limited to a maximum of 19, but *calcshares* will happily recommend values greater than this.

There is also a problem with the *nice* mechanism on AIX, present to a greater or lesser extent on all versions of UNIX. It is ineffectual when a small number of processes are being scheduled. It would be useful to be able to change some of the parameters of the standard AIX scheduler to achieve more leverage, but this is outside our own implementation guidelines.

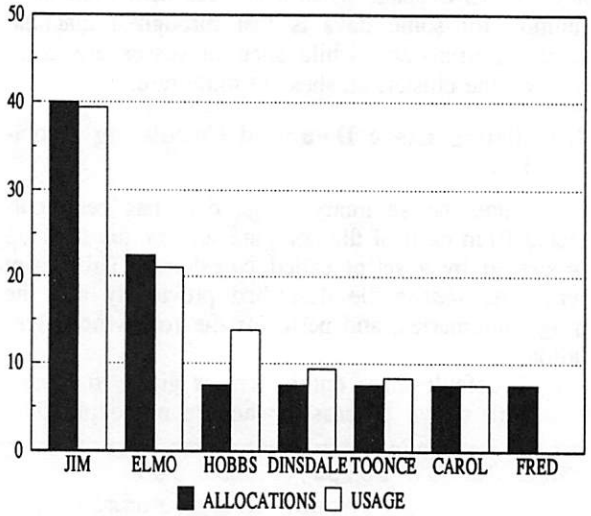


Figure 5: Comparison of allocations and usages

Data

Figure 1 is the report generated by the Watson Share Scheduler after performing its calculations.

- The %ALLOC field is how much of the machine should go to each user. Only users who are currently active, or who have been active are given a share. Fixed allocations of resources can be provided.
- The USAGE field is estimated instructions a user has executed in the time period. This is measured in tera-instructions.
- %USAGE is measured as a proportion of the total instructions delivered to users over the same period. System processes, typically less than five percent of the total usage, are ignored.
- CYCLES is number of millions of instructions per second delivered for each user over the most recent sample period (currently 5 minutes). This is so users can monitor current delivery rates.
- FAIRNESS = log(%ALLOC/%USAGE), this

measure displays how fair each user has been to other users. Negative numbers mean that a user received more instructions, whereas positive numbers mean that a user received less.

- #PROCS is the number of CPU consuming processes a user has running on the cluster. A process must accumulate 10 seconds of CPU time before it is eligible for renicing.
- NEWNICE is the priority to which the processes will be reniced.

Figure 2 is another report generated simply to show the distribution of users on the cluster. Displays a matrix presenting the cycles being delivered for each of the active users and machines on the cluster. Each machine in this example is nominally 29.9 cycles.

Figure 3 shows the usage of an individual after becoming active. The vertical axis is the proportion of the total instructions delivered to an individual user over a period of time. Leveling off occurs once the users' allocation has been reached. Note that if a users' allocation is not fixed, it can change from time to time dependent on the number of active users on the cluster (refer to the elevation in the allocation line caused by a new user becoming active). The lag between the time that the usage meets the allocation is dependent upon how heavily loaded the cluster may be, and other considerations.

Figure 4 displays the CPU utilization by users for prime shift, 9am to 5 pm. The average utilization was 94%. This does *not* include system overhead.

Figure 5 displays graphically the information obtained from Figure 1 comparing the users allocations to the percent of usage actually delivered. This indicates that the Watson Share Scheduler actually works.

Conclusion

Future changes to the Watson Share Scheduler are intended, since it has already proven to be a valuable tool. Related research has yielded an analytical model to be used to calculate the recommended nice values. The rather clumsy mechanism for transferring usage statistics to the master machine will be replaced with more efficient and flexible daemons to remove the dependence on the accounting system's processing (although the data will still be from that source). A number of illustrative graphic displays of cluster usage are being derived from the system, and are proving invaluable in managing the cluster and providing service.

Author Information

Carla Moruzzi is a Senior Associate Programmer for the Computing Systems Department at IBM's Thomas J. Watson Research Center. She has been involved with the development of the usage accounting system for the RS/6000 clusters. Carla holds a B.S. in Industrial Engineering from Clarkson University. Reach Carla at IBM T.J.Watson Research Center, 88-Z04, P.O.Box 218, Yorktown Heights, N.Y. 10598; or mail to him electronically at cmm@watson.ibm.com.

Greg Rose has been involved in the development of UNIX since it arrived in Australia in 1974. Greg has lectured in Computer Science at the University of New South Wales, which gave him an Honours Degree. He previously held the position of President of AUUG, the Australian Unix User's Group, and was Technical Director of Softway Pty Ltd., an Australian system software company. Greg is currently a visiting scientists in the Computing Systems Department at IBM's Thomas J. Watson Research Center. Reach Greg at IBM T.J.Watson Research Center, 22-136, P.O.Box 218, Yorktown Heights, N.Y. 10598; or mail to him electronically at ggr@watson.ibm.com.

Adding Additional Database Features to the Man System

Carl Shipley - Jet Propulsion Laboratory, California Institute of Technology

ABSTRACT

The on-line reference manual system is a natural way to distribute information on a large Unix network. However, the *man* system has a number of limitations which restrict its usefulness as a tool for developing databases. Some of these limitations have been corrected in recent revisions of the *man* system written in both perl and C.

This paper describes a perl script, reportman, designed to further extend the capabilities of the *man* system. The most important feature of this script is that it can be used to generate reports listing selected subsets of information from a group of manual files. The program can easily be configured to operate on different reference manual style databases through use of a database definition header file.

The program will be discussed in two example situations: maintaining a man page database of information about the configuration of individual workstations, and maintaining a database of information about third party software.

Introduction

Administration of a large Unix network involves a number of problems that are most naturally solved by the use of some kind of database. For example, the Space Flight Operations Center (SFOC) at the Jet Propulsion Laboratory (JPL) maintains a development LAN of about 100 workstations and an operational LAN that is growing to almost 200 workstations. As these networks have grown, the need has also grown for a central, easily accessible database recording the characteristics of each machine, e.g., its location, normal users, hardware configuration, operating system customizations, locally developed and third party software installed, etc.

Similarly, as the size of the network has grown, so has the number of third party software products on it, and the need for a database of information about these products. The SFOC development LAN alone now has over 110 third party software packages running on it (primarily from public domain sources) and, based on the current rate of growth, should have over 200 within two years.

In order to track information on machines and software we needed a database system that would be easily accessible and usable by managers, system administrators, and users alike - in short by everyone on the LAN. It is possible to develop a solution to this kind of database problem using DBM or any of a number of excellent commercial databases available for a Unix system. However, these approaches have drawbacks. DBM is really a set of database tools, the use of which for even a mildly complicated database system involves considerable program development. Commercial databases are generally expensive. Use of either a locally developed or a commercial database might require extensive user training.

A database system available on all Unix systems, and one that is especially attractive on a large network where information has to be shared across groups of people who may have little in common except their access to the LAN, is the on-line manual system. Although perhaps not generally viewed in this way, the on-line reference manual entries are database records, each of which consists of fairly standard fields, e.g., NAME, SYNOPSIS, DESCRIPTION, OPTIONS, etc.

The on-line manual system provides several features that are desirable for the kind of database problems discussed above. Most importantly, it is free and everyone already knows how to use it. Since it is familiar to essentially all Unix users, it is a very natural and effective way to share information on a large network. No commercial database program would be as easily accessible to an entire user population as the on-line manual system.

Toward a Re-formed man(1)

Unfortunately, the on-line manual system also has serious limitations as a database. Some of these limitations have been overcome in two fairly recent, publicly available revisions of the *man* program.

At the fourth LISA conference, Tom Christiansen [1] presented a paper describing a perl version of the *man* system that provides a number of useful extensions. For example, it allows the user to define arbitrary *man* file subdirectories, provides indexes of long manual sections, stores the whatis database in DBM format, allows typesetting of man pages, and

does automated validity checking of some aspects of the man files. In addition, a public domain *man* program written in C [2] by John Eaton, is available. This program also addresses some of the problems of the original *man* system such as the need for arbitrary file subdirectories.

However, these programs do not provide a database capability that is especially important for our applications, the need to generate simple database reports. We wanted to generate reports that answered questions like - what is the list of machine names, locations, and hardware configurations for all machines on our LAN owned by a given project and running a given piece of software.

While it is fairly easy to organize the information needed to answer this kind of question in a set of man page files, the *man* program cannot generate the report. It cannot generate a report containing information from selected files - except for the very limited case of the NAME section searched by the *man -k* and *-f* switches. It cannot do even simple sorts, as on a date, nor can it generate reports based on sorting by key words in any field, or do sorts based on combinations of keywords. These are very useful capabilities, any of which would be available in a commercial database.

The Reportman Program

The reportman program was written to overcome some of these limitations. The primary design goal was to provide the ability to do simple reports from a set of data organized as manual pages. Reportman was written in perl, a programming language developed by Larry Wall [3] that is especially suited for the kinds of text manipulation needed to extract information from a set of manual files and generate reports from this information.

The reportman program is designed to operate on a group of manual files, each of which provides information relating to the same general subject. It can be easily configured for use on any set of manual pages that conform to a few general rules. The files must be installed in the same directory, generally one of the locally configurable directories that the *man* program searches. The directory should be in either the default man directory paths or in a directory specified by the MANPATH variable or the -M man switch. Reportman's operations act on all files in the directory.

The program reads information from /usr/man/cat* files - that is, files that are in the format created by *catman*. The files must have section headers in the normal man style, i.e., as produced by the nroff macro .SH section header directive. These section headings begin in the first column and are all capitals. Information within the section does not begin in the first column. These conventions follow normal formatting produced by *catman* using the man macros. The group of files constituting a database should contain the same fields. The program will still work if the files contain extra fields, but there will be no way to include those fields in a report. It will also work if some files do not contain all the fields, although this can lead to misleading results. The program contains a utility, discussed below, that checks to see whether database files contain the appropriate fields.

Database records in a locally written reference manual style database are equivalent to the normal man sections like NAME and DESCRIPTION and can be read in the same way as other /usr/man files using the *man* program. A typical reference manual file consists of several sections, each beginning with a line containing the section name in the form, SOME SECTION HEADING. Typically, the first section is NAME.

The activity of reportman is guided by perl arrays that describe the sections in each file of a given set of reference manual files. These arrays are read from a header file, included in the program with the perl 'require' command. By default, the program looks for a header file named 'base.h' in the current directory, but the name of an alternate header file can be entered as a command line argument. The header file contains the pathname of the directory containing the manual files. Use of different header files makes it easy to use reportman with different manual file databases.

The arrays in the header file contain all the information needed to define a database. The most important of these arrays is @fields (@ is perl syntax denoting a simple array), the list of section headings used by the files in the database. This array is used to generate menus of options within reportman (e.g., the menu asking which sections to include in a report) and in pattern matches that find the desired sections within a file. A second array, %types (the % indicates an associative array, one in which strings function as array subscripts) defines the data type of the field. Reportman recognizes three types, simple text, dates, and numbers and uses these field type definitions in determining what kind of sort to do on a specified field in the sort routines. The third array, %keywds, defines suggested keywords for a given field, e.g., Sun3 for an ARCHITECTURE field in a set of manual pages describing machines on a LAN.

The program has 5 basic options which allow the user to carry out simple database configuration and report generation activities. Not all options need be enabled for a particular database. Since some options allow the user to make major changes to a group of manual files, their use would require root permissions in most cases. The set of available options is controlled by an array, %mainmenu, in the database header file.

The options are:

1. Make a new manual page.
   The program prompts the user for the file name, makes the new manual file, based on information in the database definition header file array @fields, and writes the name into the NAME field. All other fields are initialized with the string "Not assigned."
2. Add or delete a field across all manual pages in the directory.
   The program allows the user to add a new section - the content of which is initialized to "Not Assigned" - across all files in an existing database, or to delete a section across all files. This is especially helpful when developing a new database since it allows one to avoid a lot of manual editing. The program makes a backup while it is changing a database file and erases it if the change is made without error.
3. Enter information into an existing record.
   This option allows any user (with appropriate permissions) to enter information into a file via a menu-driven routine. For fields with suggested keywords, the program provides a list of these keywords.
4. Generate a report
   The report subroutine prompts the user for input about whether to save the report to a file, what sections to include in the report, and whether or not to do a sort. Any set of fields can be included in a report. The user is presented with a menu listing all the sections of the database and enters numbers from the menu indicating sections to be included in the report. If the user wants information included in the report on the basis of some sort criterion, the program determines what kinds of sorts are appropriate based on the types of data in the fields.
   Three types of sorts can be done - keyword, date, or number. The keyword sort menu provides 3 options: a simple keyword sort - true if a given keyword exists in a specified field, a sort on multiple keywords in which all must match to make the sort true, or a sort on multiple keywords for which only one must match to make the sort true. The date sort menu provides 4 options - sort all entries in ascending order, sort all entries in descending order, sort for entries before a specified date, sort for entries after a specified date. A numerical sort can be done for either ascending or descending order. It is essentially identical to the date sort.
5. Check database consistency.
   The consistency subroutine provides a menu with 3 options: check sections, check types, check percent populated. The check sections routine looks at all the files in the database and issues a report listing any files that do not contain all of the sections in the @fields array or that contain sections that are not in the @fields array. These are warnings, since the program will still work in these cases, although its output might be misinterpreted.
   The types check looks at each section in the database for which a suggested word exists in the %keywds array and reports files and sections for which it cannot find an instance of at least one of the keywords. Again, this is simply a warning. It will also look at each section defined as a date in the types array and make sure it can be parsed by the routine that sorts dates.
   The population check looks at each file in the database and determines how many of its sections still contain the "Not Assigned" phrase entered in all sections when a field is first created.

Use of the Program in Typical Applications

A Database of Machines

As machines get cheaper, we keep getting more of them. In order to keep accurate information about the hardware and software configurations of these machines, we have developed a manual page for each of our workstations. This file contains information about the location of the machine, id numbers used when scheduling a repair, the hardware configuration, the software configuration including the operating system, important third party software, and locally developed software, what JPL project is the owner and primary user of the machine, and any details of local configuration that are important for administration, like printer connections and specialized hardware.

We try to begin a manual page on a machine while it is still in the procurement stage. Of course, the machine exists in several other databases kept at JPL, but the wide distribution and ease of use of the manual page database makes it especially useful for sharing information across all the people who need to know about a workstation. In our case this involves different groups that procure the machine, check the hardware and install it when it arrives, connect it to the appropriate network, install the operating system and third party software, and finally, those who actually use it.

The reportman program allows us to sort the machines database into useful subsets. For example, machines on our LAN can be owned by any of several different administrative groups. It is easy to sort machine names, locations, etc., by group ownership using the project as a keyword. Different machines have different software products installed. Reportman can generate a list of all machines with a given piece of software by simply sorting on a

keyword in the software configuration section. In general, the program makes it easy to construct a report categorizing our machines on the basis of any word or combination of words in the machine *man* page database. The report can contain any combination of the fields in the database.

An important feature of this kind of database is that it can be at least partially populated in an automated fashion through a script, and its contents can also be periodically checked for accuracy by a similar script. Most of the information in the hardware and software configuration fields can be generated by perl - for example, machine architecture, memory size, swap space, the size of IPC parameters used in building the kernel, whether the machine has local disks, whether it is a client or a server, whether it has a particular software product installed, etc. Figure 1 presents a simple example that illustrates the general technique. The script that generates this example, shown in Figure 2, checks a few machine parameters and writes them to a file using standard man macro formatting. It uses nroff to convert the file to the normal cat file format. The output of this script run on a Sun workstation named elvis might resemble the output shown in Figure 1.

```
NAME
      elvis
LOCAL INFORMATION
      Project: Mars Observer
      Location: Not Assigned
CONFIGURATION
      sun4
SOFTWARE INSTALLED
      Fortran
      DMD
```

Figure 1: Output of the perl script

In this example, the script determines the machine name, its architecture, some information about ownership and location previously stored in a file /etc/localinfo, and whether Fortran and a locally developed program called DMD are installed. The example is artificially brief, but it illustrates basic techniques that can be used to record a more complete set of information. Since not all information one might want to save in a database can be determined using Unix commands, some data, like the room number in which the machine is located, can be stored in a text file as is done in the localinfo file in the example.

However, most information in a man file for a machine can be gathered automatically. For example, information on system configuration can be obtained by using perl to launch a C program to read kernel variables, or by reading the system messages files, or through standard commands like *pstat*. And information gathered by a script can also be periodically checked by a script, thereby eliminating much of the work of administering the database. With a completely automated system, one might simply rewrite the manual page files periodically so that when any machine's configuration was changed its manual page would be updated automatically.

```
chop ($name = 'hostname');
$localinfo = 'cat /etc/localinfo';
$ar = 'arch';
if (-f '/usr/lang/f77')
    { $hasfortran=1; }
if (-f '/usr/local/DMD')
    { $hasDMD=1; }
$out = '/usr/local/man/mano/'
                    . $name . '.o';
$putcatout = '/usr/local/man/cato/'
                    . $name . '.o';
open (OUT, ">$out);
select (OUT);
print ".TH $name o\n";
print ".SH NAME\n";
print "$name\n";
print ".SH LOCAL INFORMATION\n";
print "$localinfo\n";
print ".SH CONFIGURATION\n";
print "$ar\n";
print ".SH LOCAL SOFTWARE \n";
print $hf =
 ($hasfortran ? "Fortran\n" : "");
print $hdmd =
         ($hasDMD ? "DMD\n" : "");
close OUT;
$tmp='nroff -man $out > $putcatout';
```

Figure 2: The man page generating perl script

A Database Of Information On Third Party Software

The SFOC development network at JPL has accumulated more than 110 third party software packages from commercial and public domain sources. Recently, there has been a fairly dramatic increase in the rate at which software is acquired for the LAN, due partially to the appearance of a number of relatively inexpensive Unix products ported from the PC market, but primarily to the large and rapidly growing number of high quality software packages available on the internet. At the current rate of acquisition, the SFOC LAN will have on the order of 200 third party software products within two years. As the number of third party software packages grows, managing them has become an increasingly difficult job.

In order to help organize information in this area, we have developed a database of administrative information on third party software. In order to distribute the information as efficiently as possible, we organized the information into a set of on-line reference manual pages. We try to write a new manual page for all third party software as soon as be begin

an order, or as soon as we download something from the net. This man page, which we call the administrative man page, is not the same as the man page that often comes with third party software. The fields of the administrative man page are: NAME, VENDOR, BRIEF DESCRIPTION, DOCUMENTATION, ARCHITECTURE/OS, DATE INSTALLED, INSTALLED BY, PLACE INSTALLED, CONTACT WITHIN SFOC, USE WITHIN SFOC, LICENSE, INSTALLATION STEPS.

The use of this database has made ordering commercial software, which is generally done by purchasing personnel who may have limited understanding of the details of our installation, much easier to manage. We had been getting a lot of orders that were for the wrong machine architecture, wrong operating system, wrong graphical user interface, or wrong network configuration. The solution was to get the SAs more involved in ordering. Each week the purchasing administrator meets with a designated Unix System Administrator to go over any software products in procurement and review questions that may have come up about an order. The reportman script makes it easy to sort our database of software for products that are in procurement.

Use of the database also makes it much easier to keep up with programs acquired over the net. When a System Administrator brings in a program, he or she writes an administrative man page. With the reportman program it is easy to do a report listing programs that have been downloaded over the last few months, for example, or to do a report showing which programs have been downloaded by a particular SA.

After the product is installed, the System Administrators are responsible for continuing support, a large part of which is simply providing information about the software. Much of this information is in our administrative database, and some, like the detailed description of the software, is generally in a manual page written by the vendor or author, the location of which is in our database. Often our users can get the information themselves (and prefer to), if they know where to look. In order to make information on third party software more easily available, we maintain an 'intro' man page for the third party software database that contains a list of all products, a brief description, and the location of both the administrative and normal manual pages.

Use of Reportman with the Existing Man Pages

Although not designed for this purpose, the reportman program can be used to generate reports taken from any of the standard cat1-8 sections. For example, with reportman it is possible to generate a report of all the names of commands that have a given file in their FILES section in cat1 or a given command in their SEE ALSO section in cat8. It is useful to be able to include multiple sections in reports of this type, e.g., the NAME, SEE ALSO, and FILES sections in a report on what man files list a given command in SEE ALSO. It should be noted, however, that this kind of report can be quite slow if it involves a sort of the DESCRIPTION section of a large directory like cat1.

Man(1) and Superman(1)

The *man* system is being progressively rewritten to correct its various faults and improve its usefulness. Both the perl version of *man* by Tom Christiansen and the C version by John Eaton provide increased flexibility for the development of local on-line reference manual databases. The present project adds additional database functionality.

A further improvement would be to provide an interface for moving data between a reference manual database and commonly available commercial databases. This kind of interface would probably use a C program for accessing the database, perhaps using perl variables as command line arguments.

A reformed *man*system for generating a manual of information about the hardware and software configuration of machines on a LAN. This kind of database is of obvious use for large LANs, which are likely to become increasingly common, but it should be useful for installations of almost any size. Its construction and maintenance could be largely automated.

In general, improvements in the *man* system need not interfere with its present behavior. Additional database functions could be added to the program as command line options or as small routines like reportman. Either approach would retain the functions of the original man program, and thus would not be disruptive for existing installations. Ideally, this kind of improvement might lead to a kind of *superman(1)* system that maintains the behavior of the original man command and provides extensive additional database capabilities.

Program Availability

Information on obtaining the reportman program, brief documentation explaining its use, and a small sample database, are available from the author.

Acknowledgments

I would like to thank Ping Wei for her help on this project.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Author Information

Carl Shipley received a Ph.D. in Experimental Psychology in 1981 from UCLA where his research interests focused on the use of computers to extract information from biomedical signals such as EEG. He has worked as a software engineer at JPL since 1982 and is currently the Technical Group Leader for System Administration and Software Environment in the Space Flight Operations Center development group. He can be contacted electronically at carl@jpl-devvax.jpl.nasa.gov or by U. S. Mail at the Jet Propulsion Laboratory, M/S 301-260, 4800 Oak Grove Dr., Pasadena, Ca. 91109

References

[1] Tom Christiansen, "The Answer to All Man's Problems", *Proceedings of the Fourth Usenix Large Systems Administration Conference*, Colorado Springs, Co, Oct., 1990. The perl version of man(1) is available via anonymous ftp from convex.com.

[2] The man(1) program developed in C by John Eaton, jwe@che.utexas.edu, is distributed under the terms of the GNU copyleft.

[3] Larry Wall and Randal L. Schwartz, *Programming perl*. O'Reilly & Associates, Inc.: Sebastopol, CA, 1991.

Modules: Providing a Flexible User Environment

John L. Furlani - Sun Microsystems, Inc.

ABSTRACT

Typically users initialize their environment when they log in by setting environment information for every application they will reference during the session. The Modules package is a database and set of scripts that simplify shell initialization and let users easily modify their environment during the session.

The Modules package lessens the burden of UNIX environment maintenance while providing a mechanism for the dynamic manipulation of application environment changes as single entities. Users not familiar with the UNIX environment benefit most from the single command interface. The Modules package assists system administrators with the documentation and dissemination of information about new and changing applications.

This paper describes the motivations and concepts behind the Modules package design and implementation. It discusses the problems with modifying the traditional user environment and how the Modules package provides a solution to these problems. Both the user's and the system administrator's viewpoints are described. This paper also presents the reader with a partial implementation of the Modules package. Sample C Shell and Bourne Shell scripts are used to describe the implementation. Finally, an example login session contrasts the traditional user's environment with one that uses the Modules package.

Introduction

Typically, when users invoke a shell, the environment is initialized with the settings for every application they might access during a login session. This information is stored in a set of initialization files in each user's home directory. Over time, these files can incur numerous and relatively complex changes as applications move and new applications become available. Since each user has his own initialization files, keeping these files current with system-wide application changes becomes difficult for both the user and the system administrator. In this model, the user often makes environment changes during a login session by modifying the initialization files and then re-initializing the shell. The Modules package provides a way to simplify this process.[1]

The Modules package is a set of scripts and information files that provides a simple command interface for modifying the environment. Each module in the Modules package is a file containing the information needed to initialize the environment for a particular application (or any environment information). From a user's perspective, the package supplies a single command with multiple arguments that provides for the addition, change, and removal of application environment information. From the administrator's point of view, the environment information is documented and maintained in one location with each module encapsulating one application's information. Thus, it is easier for the system administrator to add new applications and ensure that the necessary environment for the application is correctly installed and maintained by the end users.

The Design of the Modules Package

The first section describes the motivations driving the design of the Modules package. The second section presents an overview of the design.

Design Motivations

Help alleviate the burden of UNIX environment maintenance for users.

The UNIX environment is cumbersome for even experienced users. Users not familiar with UNIX are both baffled and troubled by the complexity of environment maintenance. The Modules package attempts to ease this maintenance by encapsulating environment information and providing a single command for environment modification.

Ease the dissemination of information and documentation of new software.

The amount of information an administrator must convey about a new application can be large. Many variables may need to be changed for each

[1] Capitalized *Modules* refers to the package as a whole. References to a *module* file itself are uncapitalized. References to the *module* (1) command use italics with the man page reference.

new application. Because each module is self-documenting, the information is readily available for reference by users.

Make it simple to change the environment numerous times during the login session.

The amount of effort involved with adding environment information as applications are needed stops users from managing their environment dynamically. Using a new application requires looking up the necessary environment modifications, making the appropriate modifications, and finally invoking the command. Thus, it is usually worth initializing the environment with the information for every application when the shell is invoked. But if adding the environment for a new application is as simple as typing a single command, the balance turns in favor of adding to the environment just before accessing a new application.

Decrease dependency on servers when the applications on those servers are not needed.

When a directory is added to the search path, a dependency on the server containing this directory is added as well. If this server goes down, the user must wait for the server to return even though, in the case of an unnecessary directory, he doesn't need access to the directory to complete his work.

Manage the difficulties associated with frequently switching between different application releases.

Switching between two releases of the same software is simplified by making it easier to swap the two applications' environment information.

Design Overview

The Modules package should be simple to use. Therefore, it employs a single command interface, very similar to that of the *sccs*(1) [1] command. A single command with the ability to provide help is both easy to remember and simple to use.

For the Modules package to assist a system administrator, it should save time spent maintaining environments as well as installing and documenting the use of new applications.

The package must be flexible enough to accommodate any situation an application might require. It must meet users' needs in different ways. Some users will use the *module*(1) command to manipulate most of their environment. Other users will use it sparingly as an easy way to try new or rarely used applications. In addition, the package should permit experienced users to tailor it to their needs.

Finally, the solution must be shell-independent. The interface should be the same regardless of which shell the user chooses.

Problems with Traditional Shell Initialization

This section describes some of the difficulties with traditional shell initialization as viewed by the user and by the system administrator.

The User's Viewpoint

Maintaining shell start-up files can be difficult and frustrating for the user because he lacks interest or UNIX knowledge. Since modification is generally required when a new application is installed, maintaining start-up files can be demanding and time consuming in very dynamic or large UNIX environments.

When shell start-up files are used, the environment can become cluttered with unnecessary information. Information for every application, whether or not it will actually be referenced during the current session, is loaded into the shell. For the user to execute applications at the command line without using full pathnames, the search path could be very long.

When using an automounter[2], the system not only searches a longer path, but must remount infrequently referenced directories to search them. In addition, if a directory in the search path is on a server that goes down, the shell will hang trying to search that directory. Thus, the time needed to detect a "command not found" or to find programs toward the end of the search path is drastically increased. Shells that have path cacheing help this problem immensely, but a number of shells and users do not have or use path cacheing. It is best if a small path containing only the most used directories is set at initialization and supplemented just before a new application is used.

Switching between different versions of an application is usually difficult "on the fly." First, the user must know which environment variables to reset and then must enter the explicit shell commands to change the variables. Finally, the user must modify the search path to remove the old path and to add the new path. This is a cumbersome and time-consuming process that restricts the flexibility of changing between different software versions.

Accessing a new application is difficult when it requires a change in the user's environment. If the application is for temporary use, the user accesses the application by changing the environment in the current shell. If it's a long-term addition to the user's set of applications, the user edits the shell initialization files and the shell is re-initialized. Novice users often don't know how or don't want to know how to modify their environment to use the new application.

The System Administrator's Viewpoint

A system administrator currently announces the installation of a new application via e-mail or a note in /etc/motd. Usually, the notice contains a full description of the application and the environment variables that must be set to use the new application. Some users do not understand what they really need to do to use an application. This, in turn, causes numerous requests to a system administrator. Novice users often need a system administrator to help them modify their start-up files.

At most sites, a logfile or database is maintained containing the descriptions and quirks of each application so that the user can set up his environment to use an application. Maintaining such a logfile can be time-consuming as it can become very large.

The Modules Package Provides a Solution

The User's Viewpoint

Although shell start-up files are still necessary, maintaining them is easier with the Modules package. The user is provided with two options: modify the start-up files directly, or use the *module*(1) command to modify them.

Changing start-up files is simpler with the Modules package. The user only has to add new arguments to, or remove them from, the *module*(1) command in his start-up file to add or remove an application's environment. Or, if the user prefers, he can use the *module*(1) command to add module names to, or remove such names from, his start-up file's *module*(1) command.

A clean environment is readily maintained since the Modules package makes it easy to dynamically modify the environment. A minimum of environment information is initialized at start-up, and an application's environment is added only when needed. Response time is improved because search paths are much shorter on average.

The Modules package is optimum for the windowing environment under which many UNIX users work. For example, a user only loads the window system module during the login initialization. Then, in a shell window, the user uses the *module*(1) command to initialize the environment just prior to accessing a new application.

If the path to an application changes, the change will be masked by the Modules package. For example, a user loads a module named 'openwin' to use OpenWindows[3] even if its access path has changed. For the user, no environment or start-up file modification is required.

To switch between different releases, the user simply changes predefined modules. The old module is swapped out, and the new one is loaded in its place, even during the login session.

The Modules package will help inexperienced UNIX users manage their environment. They must learn a single command for manipulating their environment. This is opposed to having to thoroughly understand the quirks of setting a UNIX environment.

The System Administrator's Viewpoint

The Modules package eases the dissemination of information about a newly installed application: only the module name must be announced. If users want to use the new application, they use the *module*(1) command to add the announced module. Any users who don't use the Modules package can acquire the information needed to set their environment from the module file itself.

Each module is self-documenting. Users either access this information via the "module" display command or view the module file itself. In general, a logfile or database still needs to be maintained, but only the module name is listed for each application. Thus, when a user wants to use an application, the database references a module name that contains the current environment information. The user either loads this module directly or gets the environment information from the module and manually incorporates it into the environment.

Shell Wrappers and Modules

Shell wrapper scripts set up environment variables for a certain application when a command for that application is invoked. For each application, the system administrator creates a wrapper script and a symbolic link to the script for each command in the application. The Modules package can augment a wrapper script scheme or be used in place of wrapper scripts.

With wrapper scripts, users still add the directory containing the symbolic links to their search path in order to use the application. In this case, the Modules package augments the wrapper scheme by helping the user manage the search path.

One solution for managing the search path is creating a directory of symbolic links to all of the wrapper scripts. In this case, the user only adds one directory to his search path to access every application. Moving an application requires that every symbolic link for that application change. When many applications are installed, this directory can quickly become overwhelming and unmanageable. Documenting and finding programs in such a directory is difficult and often not very clean.

Modules provides the user with a lot of flexibility by differentiating between user and system module files. Like wrapper scripts, the Modules package can encapsulate environment details from the user.

The Modules Package Implementation

The Modules package has been implemented for both the C Shell and Bourne Shell dialects. This section describes some of the implementation details.

Modules Initialization

The Modules package is initialized when a user sources a site-wide accessible initialization script. This file is shell dependent and is usually done in a user's *.cshrc* or *.profile* upon each shell invocation.

This Modules initialization script defines a few environment variables. MODULESHOME is the directory containing the master module file and the master command scripts. MODULEPATH is a standard path variable that is searched to find module files. MODULESHOME should always be a part of this path. The *_loaded_modules* variable contains a space separated list of every module that has been loaded. All of these variables are exported so that the shell's children will have the same information and be able to keep track of currently loaded modules.

This initialization script sets up the *module*(1) command. This command is an alias or a function whichever the shell supports (see Figure 1).

```
##  module(1) as Function
module() {
 _module_argv="$*"
 . $MODULESHOME/.module.sh
 unset _module_argv
}
```

Figure 1: *module*(1) Command Initialization

Environment Modification

Because a process is unable to modify the environment of its parent, the Modules package sources scripts into the current shell.

The Modules package should not alter any part of the environment besides the variables documented in the *init-script* or the module files themselves. When a script is sourced into the current shell, it has the potential to change existing user variables. This "feature" permits the Modules package to work, but it presents a pitfall that the package must take into account. The main concern is the *module*(1) command because it uses a large number of variables to implement its sub-commands.

A couple of precautions have been taken to avert the possibility of variables being changed or destroyed by the *module*(1) command. The first precaution is the choice of variable names used by the *module*(1) command. All of the names are precede with an underscore and are all lowercase. The use of underscores should prevent most variable conflicts. It does, however, leave room for a user variable to be changed by the *module*(1) command. So, a check for possible variable conflicts is made when the bulk of the script is sourced. If a conflict arises, the user is notified.

The problem of changing existing variables could be eliminated by running *module*(1) as a subshell and sourcing the return values. I found this has an unacceptable response time for the problem being addressed. If users run into variable conflicts, they can set an option telling the *module*(1) command to run the script in a subshell and source the script's output (this code is not in Figure 1).

Since the Modules package is designed to abstract environment information from the user, it must be concerned with environment dependencies and conflicts between different applications and different versions of the same application. For example, the environment for two versions of the same application should not be loaded at the same time. Along the same lines, some applications are dependent upon other applications - for example, Sun's AnswerBook[4] is dependent upon OpenWindows[3]. Possible conflicts and dependencies are put into the module files themselves and detected by the *module*(1) command.

Shell Independence

The Modules package is shell-independent because the interface is independent of the currently executing shell. To ease administration, the module files are shell independent as well. Thus, only one copy of an application's environment information is maintained.

A number of functions or aliases are set up by the *module*(1) command. Each module calls these functions to accomplish specific, well defined tasks. For example, the *_set_environ* task is responsible for setting an environment variable. This is a line from an "openwin" module file.

```
_set_environ OPENWINHOME /depot/openwin
```

Here, OPENWINHOME is initialized to the value /depot/openwin[5].

System Module Files and User Module Files

Through the MODULEPATH environment variable, users can specify module directories to be searched before or after the site-wide directory. Thus, a distinction is made between "system modules" and "user modules."

System modules are maintained by the system administrator and contain the default initialization information for packages that are installed on the network. User modules, created by the user, are either derivatives of the site-wide modules or are new modules that are specific to the user's needs. This arrangement provides the user with the same power as the system-wide modules provide to easily change the environment "on the fly."

Implementation of Internal Module File Functions

This is the set of internal functions called by the module files themselves. These functions change paths and set variables, aliases, and dependencies.

_prepend_*path and _append_*path

Sometimes an application's path should be prepended to a search path. Other times it should be appended to a search path. The Modules package makes this distinction in the application's module file. The path modification functions are currently implemented to modify the PATH, MANPATH, MODULEPATH, and LD_LIBRARY_PATH environment variables. See Figure 2 for an example of how the alias and function to append the MANPATH variable are implemented.

As Bourne Shell Function:

```
_append_manpath() {
  if [ "$_rm_flag:-X" = "X" ]; then
    _rm_manpath $1
  else
    MANPATH="$MANPATH":"$1";
    export MANPATH;
  fi
}
```

As C Shell Alias:

```
alias _append_manpath
 'if($?_rm_flag) eval _rm_manpath \!:1;
 if(! $?_rm_flag) setenv MANPATH
 $MANPATH":"\!:1'
```

Figure 2: *_append_manpath* Alias and Function

_rm_*path

The *_rm_path* functions remove the directory, given as an argument, from their associated path. As with the append and prepend path functions, they're currently defined for the PATH, MANPATH, MODULEPATH, and LD_LIBRARY_PATH environment variables. Currently, *awk*(1) [1] does most of the work of removing and recreating the path.

When the user requests that a module be removed, the *_rm_flag* is set in a higher level function. Then, the module is reloaded with this flag set. Thus, every function that was called when the module was loaded is called again with the *_rm_flag* set (see Figure 2). The same functions that set up the environment now call their associated *_rm_*path* function to remove their environment information. See Figure 3 for an example of how the function to remove a directory from the MANPATH variable is implemented.

_set_environ

This function is responsible for setting and clearing environment variables. The code gets more complex when removing environment variables.

As Bourne Shell Function:

```
_rm_manpath() {
  MANPATH=`echo $MANPATH |
    awk -F: 'BEGIN {p=0}
    {
      for(i=1;i<=NF;i++)
      {
        if($i!="'$1'"){
          if(p) {printf ":"}
          p = 1;
          printf "%s", $i
        }
      }
    }'`
  export MANPATH;
}
```

Figure 3: *_rm_manpath* Function and awk Script

Since environment variables are often used to define paths to other directories, variables and paths defined later in a module must be able to reference these variables even as they are removed. The best way to describe this problem is with an example module file (see Figure 4).

```
##   OpenWindows Version 2.0
_set_environ OPENWINHOME /depot/openwin
_set_environ DISPLAY     `hostname`:0.0
_prepend_newpath  $OPENWINHOME/bin/xview
_prepend_newpath  $OPENWINHOME/bin
_prepend_manpath  $OPENWINHOME/man
_prepend_ldpath   $OPENWINHOME/lib
```

Figure 4: OpenWindows module File

Here the first variable set is the location of OpenWindows (OPENWINHOME). This variable is used to define the path locations as well. If the variable were removed from the environment before the paths were removed, it would be impossible to remove the paths. So, environment variables are not actually cleared from the environment until after the module has been removed. Thus, *_set_environ* simply adds any environment variables to a list. Upon completion of reading the module file, the elements are removed from the environment. See Figure 5 for an example of how the alias and function defined to set and unset environment variables are implemented.

_prereq and _conflict

Two functions were created to manage conflicts and dependencies with other module files. The *_prereq* function is a list of modules the calling module must have loaded to run. Similarly, the

_conflict function is a list of modules the calling module has conflicts with. If more than one module is listed for these commands, it is treated as an ORed list. Multiple calls can be used to get an ANDing effect.

Bourne Shell Function

```
_set_environ() {
  if [ "$_rm_flag:-X" = "X" ]; then
    _unset_list="$_unset_list $1"
  else
    eval $1="$2"; export $1;
  fi
}
```

C Shell Alias

```
alias _set_environ '
  if($?_rm_flag)
    set _unsetenv_list = \
          ($_unsetenv_list \!:1);
  if(! $?_rm_flag)
    setenv \!:1 \!:2'
```

Figure 5: *_set_environ* Function and Alias

For example, AnswerBook needs OpenWindows loaded to run. So, it defines the openwin (or openwin-v3) module as a prerequisite. A conflicting case would be OpenWindows Version 2.0 with OpenWindows Version 3.0. These two modules should not be loaded at the same time. So, each one defines the other as a conflict. See Figure 6 for an example of how the function for conflict management is defined.

```
_conflict() {
  if [ "$_rm_flag:-X" = "X"]; then
    return;
  fi

  _conflict_loaded=0
  for _con in $*; do
    for _mod in $_loaded_modules; do
      if [ "$_mod" = "$_con" ]; then
        _conflict_loaded=$_mod
      fi
    done
  done
  if [ $_conflict_loaded != 0 ]; then
     echo "ERROR: Module conflict"
  fi
  unset _conflict_loaded
}
```

Figure 6: *_conflict* Function

Implementation of the module(1) Command

Each invocation of *module*(1) sources a site-wide script that actually implements the command. Arguments are passed to the *module*(1) command using the *_module_argv* variable (see Figure 1).

The first argument designates the sub-command the *module*(1) command is to execute. Valid arguments are as follows:

- Load or Add
- Remove or Erase
- Switch or Change
- Show or Display
- Initadd
- Initrm
- List
- Available
- Help

Loading Modules

Loading modules is done by the *_add_module()* internal function. This function takes any number of arguments and attempts to load each one as a module. It traverses the argument list, first verifying that a listed module isn't loaded already. If it is, the function prints an error and moves on to the next name in the argument list (see Figure 7).

```
_add_module() {
if [ $# -lt 1 ]; then
 echo "ERROR: More arguments"
 return
fi
for mod in $*; do
  _found=0; _cur_module=$mod;
  for chkmod in $_loaded_modules; do
  if [ $mod = $chkmod ]; then
    _found=1
    break
  fi
done
if [ $_found -eq 1 ]; then
 echo "ERROR: Module already loaded"
 continue
fi
```

Figure 7: *_add_module()* Argument Verification

If the module is not loaded, *_add_module()* begins looking for the module by searching each directory specified in the MODULEPATH variable. Once found, the module is sourced, the module name is appended to the *_loaded_modules* variable, and if there are any other arguments, the load process begins anew (see Figure 8).

Removing Modules

Removing a module is accomplished by using the *_rm_module()* internal function, which is very similar to the *_add_module()* function. Any number of arguments can be passed to the *_rm_module()* function. First, each argument is checked to verify that the module is actually loaded (same code as in Figure 7 except the final if statement indicates an error if the module is not loaded). If a module is

loaded, the *_rm_flag* is set and the MODULEPATH variable is searched. Once a module is found, it is sourced (see Figure 9).

```
for dir in $MODULEPATH; do
  if [ -f $dir/$mod ]; then
    echo "Loading $dir/$mod"
    . $dir/$mod
    if [ "${_load_error:-NotSet" =
                  "NotSet" ]; then
     _loaded_modules=
         "$_loaded_modules $mod"
     export _loaded_modules
     unset _load_error
    fi
    _found=1
    break
  fi
done

if [ $_found -ne 1 ]; then
  echo "ERROR: Module not found"
fi
```

Figure 8: *_add_module()* Traverses MODULEPATH and Loads Module

```
_rm_flag=
  for dir in $MODULEPATH; do
   if [ -f $dir/$mod ]; then
    echo "Removing $dir/$mod"
    . $dir/$mod
    _loaded_modules=
      `echo $_loaded_modules | \
              sed s/$mod//`
    export _loaded_modules
    _found=1
    break
  fi
 done
  if [ $_found -ne 1 ]; then
    echo "ERROR: Module not found"
  fi
  for env in $_unset_list; do
    unset $env
  done
```

Figure 9: *_rm_module()* Traverses MODULEPATH and Loads Module

The same functions that are used in loading a module are used to remove modules. When one of the functions detects that the *_rm_flag* is set, it removes its corresponding piece of environment instead of adding it (see Figures 2 and 3). Note that after the module file has been sourced, each variable in the *_unset_list* is unset.

Switching Modules

Although this function has not been fully implemented, I will describe it here. Only modules that define themselves as compatible with another module can be switched. The compatibility information is kept in the module file. If a module can be switched with another module, it lists that other module via the *_switch()* function.

Switchable modules are very similar in that their environments match one another and their modules follow the same format. The variables that have to be reset are the same, and the search path changes are the same as well.

The difference between switching two modules and the process of removing a loaded one and loading a new one is that the location of a search path entry does not change. The append and prepend functions are used when removing and loading module files. This process has the possibility of altering a portion of the search path in relation to other entries.

Although this sounds restrictive, it is often very useful because most module switching involves different versions of the same program.

Displaying Modules

The *_display_module* internal function implements the user display and show sub-commands. If no arguments are provided, it displays information about every loaded module. Otherwise, only the modules named as arguments are displayed. An *awk*(1) script is used to convert the information contained in the module file to output that is visually pleasing to the user.

Changing User Initialization Files

The initadd and initrm sub-commands help the user add modules to and remove modules from their shell initialization files. The initialization file is searched for a comment line placed there by the *module*(1) command. Located immediately after this comment line is a line invoking the *module*(1) command. It is this line that is changed according to the list of modules given to the *initadd* or *initrm* request. If a comment line is not located in the initialization file and the user is requesting that a module be added, the comment line and *module*(1) command line is appended to the user's initialization file.

Listing Modules

The *_list_modules()* function simply prints out the current value of the *_loaded_modules* variable.

Available Modules

Each directory in the MODULEPATH variable is listed using the UNIX *ls*(1) [1] command by the *_avail_modules()* function (see Figure 10).

```
_list_modules() {
  if [ "$_loaded_modules" ="" ]; then
    echo "No Modules Loaded"
  else
    echo "Loaded: $_loaded_modules"
  fi
}
_avail_modules() {
  for dir in $MODULEPATH; do
    echo $dir":"
    (cd $dir; ls)
  done
}
```

Figure 10: *_list_modules()* and *_avail_modules()* Functions

Help for Modules

Two levels of help are provided. Without any arguments, the *module*(1) command lists the available sub-commands. With 'help' as the only argument, it provides more complete description of the Modules package. If a second argument, the name of a sub-command, is provided, the *module*(1) command displays help about the sub-command.

Example Sessions

Contrast Conventional Style with Modules Style

The examples in Figures 11 and 12 depict how the same environment modifications are accomplished with and without using the Modules package. Specifically, the examples show how a user would switch from using OpenWindows Version 2.0 to a Development Version of OpenWindows Version 3.0. The keystrokes the user actually types are in bold. Notice the difference in effort needed to switch between the two window systems. Also notice the difference in the length of the search path variables.

```
system login: jlf
++++++++ CSH Login ++++++++
jlf@system% echo $PATH
/depot/lang:/depot/openwin/bin:
/depot/openwin/bin/xview:/usr/local/bin:
/usr/bin:/usr/ucb:/usr/etc:.:
/depot/frame/bin:/depot/sunvision/bin:
/depot/TeX/bin
jlf@system% setenv OPENWINHOME /depot/openwin-v3
jlf@system% setenv PATH /depot/lang:
/depot/openwin-v3/bin:/depot/openwin-v3/bin/xview:
/usr/local/bin:/usr/bin:/usr/ucb:/usr/etc:.:
/depot/frame/bin:/depot/sunvision/bin:
/depot/TeX/bin
jlf@system% setenv LD_LIBRARY_PATH
/depot/lang/SC1.0:/depot/openwin-v3/lib:/usr/lib
jlf@system% setenv MANPATH /depot/lang/man:
/depot/openwin-v3/man:/depot/sunvision/man:
/depot/TeX/man
jlf@system% openwin
```

Figure 11: Conventional Style

```
system login: jlf
++++++++ CSH Login ++++++++
Loading /site/Modules/openwin
jlf@system% echo $PATH
/depot/openwin/bin:/depot/openwin/bin/xview:
/usr/local/bin:/usr/bin:/usr/ucb:/usr/etc:.
jlf@system% module rm openwin
Removing /site/Modules/openwin
jlf@system% module add openwin-v3
Loading /site/Modules/openwin-v3
jlf@system% openwin
```

Figure 12: Modules Style

A More Complex Modules Example

A few more *module*(1) commands are demonstrated in Figure 13. In this example, after logging in, the user checks what modules are currently available. Then, the 'lang' module is displayed to find out what the module does. Notice that the PATH environment variable is changed as the module display indicates. The 'answerbook' module is displayed showing how prerequisites might be used. In this example, answerbook must have either the 'openwin' or 'openwin-v3' module loaded before it will load. Finally, the 'answerbook' module is loaded and the program is started.

```
system login: jlf
++++++++ CSH Login ++++++++
Loading /site/Modules/openwin
jlf@system% module avail
/site/Modules:
X11@          init-csh openwin      tex
X11R4         init-sh  openwin-v3   vx-devel
answerbook    lang     saber        xgl
dos           local    sunvision
frame         lotus    sunvision-devel
frame-ol      mh       taac-devel
jlf@system% module show lang
++++++++ ( /site/Modules/lang Module ) ++++++++
Unbundled Languages
Prepend PATH: /depot/lang
Prepend MANPATH: /depot/lang/man
Prepend LD_LIBRARY_PATH: /depot/lang/SC1.0
jlf@system% echo $PATH
/depot/openwin/bin:/depot/openwin/bin/xview:
/usr/local/bin:/usr/bin:/usr/ucb:/usr/etc:.
jlf@system% module add lang
Loading /site/Modules/lang
jlf@system% echo $PATH
/depot/lang:/depot/openwin/bin:
/depot/openwin/bin/xview:/usr/local/bin:
/usr/bin:/usr/ucb:/usr/etc:.
jlf@system% module show answerbook
+++++ ( /site/Modules/answerbook Module ) +++++
Answerbook Version 1.0
Prerequisites(ORed): openwin openwin-v3
Append PATH: /depot/answerbook
jlf@system% module add answerbook
Loading /site/Modules/answerbook
jlf@system% answerbook
```

Figure 13: Complex Modules Example

Future Work

Having the Modules package as a set of scripts that are sourced into an existing shell helps makes the interface shell independent. However, it would be best for performance and cleanliness to have support for the Modules package built into the shell itself. The module commands could be more complex without losing any performance over the current version.

Currently, conventional style search paths can be built in an order that doesn't represent a logical search structure. Users are responsible for maintaining the order of their paths with little or no help. The Modules package can provide the user with the information he needs to build a logical search path with existing modules.

Work is in progress to increase the grammar available in the module file. Syntax permitting if-else statements and path ordering are two examples of new syntax not described in this paper.

Finally, more options are under development to provide users with even greater control over how their environment is constructed by the Modules package. In some cases, users don't want a variable modified when loading a certain module file. For example, if the OpenWindows[3] dynamic libraries are already cached using *ldconfig* (8) [1] the LD_LIBRARY_PATH variable should not be modified when loading the "openwin" module file. Other configuration and control options are being added for more experienced users.

Results and Performance Notes

The Modules package is quite new and is still under development. Only a few of our users have begun working in the Modules environment. They have been very pleased with the package and the benefits it provides.

Currently, it takes a second or two to load a module into the current shell. In an effort to improve this performance, it is possible for the user to specify that the internal functions remain resident from one *module* (1) command to the next. This provides a marked improvement in speed since the functions are not being completely redefined upon every invocation.

Summary

The Modules package provides both the novice and the experienced UNIX user with a clean interface to the environment. This interface enables the user to easily add, change, and remove application environments dynamically.

Author Information

John L. Furlani graduated from the University of South Carolina with a BS in Electrical and Computer Engineering. He worked as a system administrator at both USC and the Naval Research Laboratory in Washington, D.C. during his college years. Upon graduation, John joined Sun Microsystems Incorporated as the system administrator for Sun's Research Triangle Park Facility. Reach him at Sun via U.S. Mail at Sun Microsystems Incorporated, P.O. Box 13447, Research Triangle Park, NC 27709-13447. Reach him via electronic mail at sun!sunpix!jlf@uunet.uu.net or via the internet at john.furlani@East.Sun.COM.

References

[1] Sun Microsystems Incorporated, *SunOS Reference Manual.*

[2] Sun Microsystems Incorporated, "Using the NFS Automounter", *System and Network Administration*, Chapter 15.

[3] Sun Microsystems Incorporated, *OpenWindows Version 2 Reference Manual.*

[4] Sun Microsystems Incorporated, *Using the Sun System Software Answerbook.*

[5] Manheimer, Warsaw, Clark, Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries", *USENIX Large Installation System Administration IV Conference Proceedings*, October 1990, p. 37-46.

NAME

module - command interface to the Modules package

SYNOPSIS

module [*sub-command*] [*modulefile* ...]

DESCRIPTION

The **module** command is the user interface to the Modules package. The Modules package provides for the dynamic modification of a user's environment via encapsulated *modulefiles*.

Each *modulefile* contains the information needed to initialize the environment for a single application or working environment. Once the Modules package is initialized, the environment can be modified on a per-module basis using the **module** command. The modules are added to and removed from the current environment by the user. The environment information contained in a *modulefile* can be summarized through the **module** command as well. If no arguments are given a summary of the usage and sub-commands is provided.

Initialization

The Modules package and the module command are initialized when a shell-specific initialization script is sourced into the shell. The script is named **init-***shellname*. For the C Shell, the **init-***csh* script is sourced. The master module directory containing all of the module files is /usr/lib/Modules.

Sub-Commands

The second argument designates the action the module command will take. A summary of these commands are as follows:

load, add	Load the given *modulefile(s)*.
erase, rm	Remove the given *modulefile(s)*.
switch, change	Switch the two related *modulefiles*.
show, display	Display information about each given *modulefile*.
initadd	Add the given *modulefile(s)* to the user's shell initialization file.
initrm	Remove the given *modulefile(s)* from the user's shell initialization file.
avail	List all available *modulefiles* in MODULEPATH.
help	Provides more details about the command.

Modulefiles

A *modulefile* contains the environment information for a single application or working environment. There are a number of functions for modifying the environment that each *modulefile* uses. These functions are as follows:

_set_environ *variable value*
Sets *valiable* equal to *value*. Similar to the csh setenv command.

_append_manpath *value*
_append_newpath *value*
_append_ldpath *value*
_append_modulepath *value*
Appends *value* to the associated environment path variable. The actual variables are MANPATH, NEWPATH, LD_LIBRARY_PATH, and MODULEPATH repectively. Please note that the more generic **_append_path** function obsoletes these.

_append_path *variable value*
Appends *value* to the path syntax environment variable *variable*. This function obsoletes the older set of variable specific functions.

_prepend_manpath *value*
_prepend_newpath *value*
_prepend_ldpath *value*
_prepend_modulepath *value*
_prepend_path *variable value*
Performs the same function as their append counterparts except *value* is prepended to the variable.

_rm_manpath *value*
_rm_newpath *value*
_rm_ldpath *value*
_rm_modulepath *value*
_rm_path *variable value*
Removes *value* from the associated path variable. Again, **_rm_path** obsoletes all of the other variable specific functions.

_prereq *modulefile* [*modulefile* ...]
_conflict *modulefile* [*modulefile* ...]
In the case of **_prereq,** the arguments are a list of *modulefiles* that must be resident before the loading of the current *modulefile* will continue. In the case of **_conflict,** the arguments are a list of *modulesfiles* that this *modulefile conflicts with and must NOT* loading of the current *modulefile* will continue. This is usually put immediately after the comments in a *modulefile* to inhibit loading the *modulefile* if dependencies are not met. Note that when multiple arguments are given to the **_prereq** or **_conflict** function, the effect is a logical ORing of the listed *modulefiles*. To achieve a logical ANDing effect, use multple **_prereq** or **_conflict** calls in succession.

ENVIRONMENT

The behavior of **module** can be tailored by means of the following environment variables:

MODULEPATH
This is the path that **module** searches when looking for *modulefiles*. It is set to the master module directory (/usr/lib/Modules) at initialization. This can be set to search group module directories or individual module directories before or after the master module directory.

_module_keep_functions_resident
When this is set, **module** does not remove the internal functions from the environment when it is finished. This helps to speed up successive **module** invocations because the functions do not get redefined each time. It is the default for **module** to clean up after each invocation.

_module_no_manpath
_module_no_ld_library_path
This is a list of *modulefiles* that should not modify the MANPATH or the LD_LIBRARY_PATH environment variables respectively. If a *modulefile* is being loaded that is on this list, the respective environment variable will not be modified. As with the path modification functions, these are obsoleted by the more generic _module_no_modify environment variable.

_module_no_modify
This provides the same functionality as the more specific non-modify variables listed above

except the variables not to modify are specified on the list. The format for this is a colon separated list of space separated lists. Such that the first element in the list and each value immediately after a colon are the environment variables not to be modified. Each element after this element (the environment variable) until the next colon or the end of the line is a *modulefile* that should not modify the specified environment variable.

The following environment variables are set upon initialization of the Modules package and are maintained throughout. These variables are exported as to keep track of previously added *modulefiles* and previous changes in the Modules environment.

MODULES

This is the location of the master module file directory. It is used in setting MODULEPATH. If it has already been set, it will not be reset.

MODULEPATH

If this variable has not already been set by the user, it is initialized to the master module directory.

_loaded_modules

A space separated list of all the *modulefiles* currently loaded. Since this is exported, it maintains the list from the beginning of the login session.

NEWPATH

This is currently used to cache changes to the PATH environment variable or path shell variable in the case of the C Shell. In some shells that maintain hash tables, this helps improve speed because these variables are monitored for change and get rehashed upon each modification. The benefits appear to be minimal in most cases so this variable will be obsoleted and the PATH variable will be directly modified. If upon **module** invocation, NEWPATH does not equal PATH it is reinitialized with the value from PATH.

FILES

/usr/lib/Modules

This is the default directory for system *modulefiles. The location of this directory* the MODULE environment variable as described above.

/usr/lib/Modules/.module-*shellname*

This file is the script that gets sourced upon each invocation of **module.**

/usr/lib/Modules/init-*shellname*

This is the initialization file that is sourced to initialize the Modules package.

Configurable User Documentation -or- How I Came to Write a Language with a Future Conditional

Mark A. Verber - The Ohio State University
Elizabeth D. Zwicky - SRI International

ABSTRACT

We have found that the available documentation in the form of `man` pages, vendor supplied documents, and retail books is inadequate for many user's needs, especially for the naïve user. This paper describes our attempt to create a configurable user's guide which meets our users' needs while being maintainable and sharable between sites.

Our user's guide has been written in a way that permits general concepts and site-specific details to appear side by side in the guide while at the same time permitting very different sites to share the text. To facilitate the customization of the user's guide we created a small text preprocessor called tpp which is used very much the same way `cpp` is used in a C program.

The combination of macro definitions, include files, and tpp commands allowed us to create a configurable user's guide that could be shared between sites with minimal work. We have also found a number of other uses for tpp since its creation.

Introduction

In 1984 we decided that the Ohio State University Computer and Information Science department (OSU-CIS) needed an introduction to the department computing facilities. We wanted a document that pulled together all the information that a new graduate student or faculty member might need: site policies, electronic mail addresses, a list of facilities provided, and basic introductions to the various operating systems the department supported. Creating such a document was particularly important since there were no good introductions to our two primary platforms: a DECsystem-20 running TOPS-20, and a VAX running BSD UNIX.

Much of this important information was getting passed student to student in the form of oral traditions. This is fine, except that information was not spread around evenly (you needed to know one of the system gurus or you were missing important information), and once oral traditions were started it was almost impossible to kill them, even when the information became incorrect or irrelevant.

Being firm believers in not working any harder than we needed to, we looked around for a suitable existing document that we could take and adapt to our local requirements. We found a document which had been written for the Computer Science department at Carnegie-Mellon. Years went by, and various people at OSU-CIS edited the document, and edited the document, and edited the document. It grew to be three times its original size, changed text formatter, developed its own layout, grew an annotated bibliography, added its own font, and otherwise consumed time and resources.

And then the two of us who had maintained it at OSU-CIS left, to go to other jobs. Lo and behold, the new sites we were at had no equivalent user documentation. But the old OSU-CIS Facilities Guide didn't quite work for them. On the other hand, there was a not a chance that either of us was going to throw it out and start over, having invested several years into it already. Therefore, in a spirit of enlightened self-interest, we set out to build it into a user-configurable book - something that we could use at both our sites, take to new sites, and let other people use as well.

The Problem

We found the available documentation inadequate for our users. There are three main types of documentation that you can get without writing it yourself; manual pages, other manufacturer's documentation, and books.

Manual pages are not useful for truly naïve users. They don't have the right sort of information (you can't find out what your e-mail address is, or what editor you should use); the information they do have is often in words that new users don't understand; and they do have all sorts of irrelevant information that acts as noise.

Next is the manufacturers documentation. How good the manufacturer's documentation is varies; unfortunately, the range is from "almost good enough" to "you're not certain whether to laugh or

to cry''[1]. Even the best quality documentation from manufacturers tends to fall short of what the average user needs.

Most often the manufacturer provides a **very** simplistic ''getting started'' guide, and then a stack of documentation which includes all the `man` pages in printed form and stacks of detailed manuals describing major software systems such as `troff`, program development tools, etc. The result is that to accumulate enough manufacturer-supplied documentation to introduce a new user to a system generally results in a foot-high stack - the ''getting started'' guide is never enough. The new user's response to the foot-high stack is to put it in a corner and whimper, never looking at the documentation because it is too overwhelming. Furthermore, the site that uses only and exactly the software that their hardware manufacturer supplies has yet to be found.

Because of these well-understood problems with vendor documentation, you can go to your neighborhood bookstore and buy an introduction to UNIX. These days, you can even buy an introduction to Berkeley and Berkeley-derived versions of UNIX. Unfortunately, the quality of these introductions vary as much as the manufacturer-supplied documentation. Even the best of the books will tell you how to use only and exactly what most hardware manufacturers supply. For instance, it will almost certainly tell you how to use `vi` and `troff`, which is all very well if that's what your users are supposed to prefer. If your standard is `emacs` and LaTeX, or FrameMaker, these books will merely confuse your users. By the time you have put together the documentation that lists all the ways your site is different from what the book says, not only have you done as much work as it would have taken to write the documentation from scratch, but you are also back to the foot-high stack of documentation.

Goals

We wanted a document that would meet the following goals:

1. Be non-threatening: no foot-high stacks.
2. Be easy to use: use all possible tricks to make information findable.
3. Provide a link to further documentation: cross-reference to books, manufacturer's documentation and manual pages as much as possible.
4. Integrate local and general information: don't tell someone how the printer system works in one place, and what the names of the printers are somewhere else.
5. Be sufficient: provide all the necessary information to get a task done.
6. Re-use information: never require an administrator to fix information more than one place.
7. Be flexible: allow the production of more than one document, using the same information.
8. Be portable: let the same basic document be useful at multiple sites, and at the same site over time.

Our Solution

Many of these goals were met by the OSU-CIS Facilities Guide; it had most of the information in it, the writing style was accessible, there was a good index, it was fanatically structured, and it had a good bibliography. In most cases, it already had macros in place to avoid requiring multiple changes of the same information, and it even had some macros set to allow chapters or sections to be used as stand-alone documents. Unfortunately, it failed pretty badly on portability. What portability it had was the result of adapting to changes in the department's configuration over the years; there had never been any hesitation about coding assumptions about being a university in the middle of Ohio, for instance.

The changes that had already been made were in the form of LaTeX macros. After the year in which the staff offices moved 4 times (and some staff members moved 12), all the names and addresses had been split out into a configuration file, for instance. We kept those, and added to them where appropriate.

Simple string replacement was not enough to fix all the changeable parts of the document, however, and we had to add two further methods of customizing it. The first was taking text that was clearly unsalvageable - so changeable that it was going to have to be rewritten for every site - and isolating it in individual files. Examples of what we put into separate files include site policies for accounts and user behavior; descriptions of printing devices; dialup information; and tables showing supported programming language. All of these files go in a `Local` subdirectory, and examples are provided with the document. Pulling the files into the main document is done with normal LaTeX input primitives.

For other parts of the document, we needed a solution intermediate between string replacement and file insertion; in fact, we needed something like conditional compilation, where you could have the appearance of text depend on the value of variables. This could be very simple, like omitting the VMS chapter for sites that don't have VMS, or considerably more complex. Unfortunately, tools for text production don't provide features like con-

[1] The manufacturer that shipped an installation manual and a third-generation photocopy of the Berkeley 4.2 docs - complete with font tables for Berkeley's Versatec printer - falls into the latter category.

ditional compilation, except to a limited extent[2]. TeX conditionals really are not well suited for trying to comment out entire blocks of code; it's possible, but it isn't pretty.

Tools that are intended for programs do conditionals well, but break other things. For instance, `cpp` leaves blank lines where all its directives are, which is lovely for preserving line numbering, but really upsets life when blank lines change the semantics of the language - as they do in TeX and `troff`. Furthermore, it's highly inconvenient to have `#defined` variables interpreted wherever they occur in the text. Capitalization may suffice to distinguish between a pre-processor directive and a language element in C, but in English you don't get free choice of how to capitalize things, and you end up making outrageously ugly `#defines` to avoid having your text diced. (Instead of using `#ifdef UNIX`, you must use something like `#ifdef UNIXP`. Or `ifdef unix` - after all, "unix" is not legal in text unless it happens to be the name of a UNIX command - but that really upsets people who are accustomed to C.)

Furthermore, there are some structures in English that are not well handled by "if" and "if not". Using `cpp` to try to control a sentence that is supposed to end up saying something like "There are three operating systems you can make your home on; UNIX, TOPS-20, and VMS" when you don't know how many operating systems will be involved is very, very, nasty. When I tried it, I ended up with a screen full of intricately nested ifs, and a bad headache.

Our solution to this was to build a text preprocessor, which we call tpp. tpp carefully gets rid of lots of the features of `cpp`, and introduces a few new ones instead. The future conditionals mentioned in the title are among them; in sentences like the one above, you can create a case statement that fixes the "are three" to "is one" or "are many" depending on how many things are going to be true when you get to the end of the sentence.

TPP

Tpp is currently a very small language, containing a whole 7 directives. It is implemented as a perl program. The 7 tpp directives are `define`, `undef`, `if`, `ifndef`, `conj`, `number` and `bynumber`. Tpp currently considers any line beginning `%#` to be information for it; it pays no attention to other lines, and either emits them unchanged or doesn't emit them at all. `%#` was chosen for LaTeX's benefit, to allow a tpp document to be processed by LaTeX without first having been run through tpp. Future versions of tpp will allow you to choose an attention sequence.

In the current version of tpp, variables are either set or unset; they don't take values. By default, they are unset; it is perfectly legal to unset a variable that is already unset, set one that's already set, or reference one that was never explicitly set or unset. `if` and `ifndef` control emission of text, and have matching `else` and `endif`. The following code produces "Perl is fun." as its only output:

```
%#define fun
%#if fun
Perl is fun.
%#else
Perl is not fun.
%#endif
```

`conj` is the conjunction statement, used to output lists in English; it takes a conjunction as an argument, and then cases by variable.

```
%#define mares
%#define does
%#define lambs
It's true;
%#conj and
%#case mares
mares eat oats
%#case does
does eat oats
%#case lambs
little lambs eat ivy
%#endconj
```

produces "It's true; mares eat oats, does eat oats, and little lambs eat ivy." If "mares" and "does" are unset, the sentence reads "It's true; little lambs eat ivy." This may seem unimpressive, until you try to produce this effect with only `cpp` directives.

`number` takes `last` or `next` as an argument, and returns the number of cases that were true in the most recent `conj`, or are going to be true in the next one. This allows you to say

```
There are
%#number next
main weapons of the Spanish
Inquisition:
%#conj and
%#case fear
fear
%#case surprise
surprise
%#case devotion
a fanatical devotion to the pope
%#endconj
```

[2]Systems like FrameMaker and Interleaf that are used to write documentation provide some conditional facilities; aside from the limitations of those facilities, using either one to write portable documentation would be like using ADA to write portable software for UNIX workstations.

and not have the usual problem with getting the number right.

`bynumber` also takes `last` or `next` as an argument, but it then cases on the result. It allows ranges, and also the use of the keyword `many` to mean "a bigger number than I've got a case for", allowing

```
There
%#bynumber next
%#case 0
are no weapons
%#case 1-3
are a few main weapons
%#case 4-10
are several main weapons
%#case many
are lots and lots of main weapons
%#endnumber
of the Spanish Inquisition.
```

Guide Contents

Our first goal was for the Facilities Guide to be non-threatening. One part of being non-threatening was to keeping the Facilities Guide down to a reasonable size without a novice user needing other reference material. Unfortunately, that's only so achievable; the Facilities Guide configured for the Ohio State Physics department produces 220 pages of text. This is enough to be scary to a new user. We try to ease these concerns with our introduction which tells a new users that they don't have to learn everything that is contained in the Facilities Guide. The first chapter is a roadmap, which explains what each of the chapters is about and instructs a new (or experienced) user what chapters should be helpful.

We wrote the second chapter of the Facilities Guide for the computer neophyte: someone who had no idea what a text editor was, much less why using electronic mail is useful. Having this introduction helps the rest of the Guide be more useful since we could assume a basic level of understanding. The first chapter advises experienced users to skip or skim this chapter, since they presumably do not want to be told what a text formatter does and why you would want to do it.

The middle chapters of the guide address specific operating systems (currently these are UNIX, VMS, Tops-20, and the Macintosh OS, although an MS-DOS chapter is in progress). Each of the operating system chapters follows the same basic outline (as does the introductory chapter), although the details change from operating system to operating system. The hope is that the information comes in the order that people need it. Roughly, the structure is:

1. How to get in.
2. How to change your password.
3. How to get out.
4. How to give commands.
5. How to use electronic mail (including the information about what your mail address is.)
6. The file system; how it is arranged, and how you deal with files.
7. Text editing.
8. Formatting systems.
9. Printers.
10. Programming languages and tools.
11. Other topics.
12. Games.

In some cases, not all the useful information will fit into this relatively rigid structure. The UNIX chapter, for instance, has long since grown into multiple chapters, with a separate chapter to deal with window systems. Where possible, these additional chapters echo the same structure (for instance, the description of each window system starts by telling you how to start it up, how to shut it down, and how to get help).

In order to keep the operating system chapters relatively short and non-repetitive, information about significant tools that occur on multiple operating systems is pulled out into separate chapters. For instance, a separate text formatting chapter is provided with the information about LaTeX. The operating system chapters provide the operating system dependent information (how to start the executable, how to print a dvi file, where to find sample documents) and then refer to the specific chapter.

We have created an extensive index and a detailed table of contents, and a large annotated bibliography. The layout of the Facilities Guide is designed to make specific sections easy to find by using a lot of white space and horizontal rules.

The use of tpp and various TeX primitives permits us to intersperse site-specific information within the body of a very generalized section. Examples given in the text use host names and the command line prompts that are set locally. Rather than talking in general about printer support, we can say what printer is preferred for high quality output, and which printer is the fastest. In the section discussing remote access we can tell users what phone numbers to use and what they need to type to gain access to our machine via a dialup line.

New Problems We Have Created

The version of the facility guide I am looking at now takes up nearly three megabytes of disk; that's built for my configuration as far as LaTeX source. That does not include either LaTeX or perl, which work together to create the manual. Along with the actual text, the manual distribution

contains not only tpp, but also an indexing program and, believe it or not, a PostScript font - the font is only actually required for the Macintosh chapter.

All in all, it's a lot of trouble to go to for a manual, especially when you consider that you may still end up creating sections for programs or operating systems that we don't run. On the other hand, it's a lot less trouble than writing your own manual from scratch, and we will happily merge other people's sections in, with credit. (The acknowledgments section is now well into its second page.)

New Solutions We Have Created

As it turns out, there are programs besides LaTeX and `troff` that care deeply about empty lines and may have almost anything in them - `sendmail`, for instance. While the more esoteric text processing features of tpp are out of place in a sendmail.cf, the ability to do `cpp`-style preprocessing to create multiple sendmail.cfs out of a single master can be extremely handy at a large site, and in fact OSU-CIS, which is not presently using the facilities guide, is using tpp for that purpose.

Availability

Current releases of the Facilities Guide and tpp are available for anonymous ftp in the `pub` directory of the machine `ftp.solutions.com` as the files `facilities-guide.tar.Z` and `tpp.shar` respectively.

Author Information

Mark Verber is a system programmer for the Physics Department at The Ohio State University. He discovered UNIX in 1978 as a high school student and has been working for OSU since 1980. Reach him via U.S. Mail at The Ohio State University; Physics Department; 174 W. 18th Avenue; Columbus, Ohio 43210. Reach him via electronic mail at verber@pacific.mps.ohio-state.edu or verber@solutions.com.

Elizabeth Zwicky is a system administrator for the Information, Telecommunications, and Automation Division at SRI International, where she is known for writing peculiar programs in languages beginning with the letter "P". Reach her via U.S. Mail at SRI International; 333 Ravenswood Avenue; Menlo Park, CA 94025. Reach her electronically at zwicky@erg.sri.com.

We Have Met the Enemy, An Informal Survey of Policy Practices in the Internetworked Community

Bud Howell - MTEK International, Inc.
Bjorn Satdeva - /sys/admin, inc.

ABSTRACT

This survey suggests that the system administrator assumes the dominant role in both the definition and enforcement of policy at the majority of educational and commercial sites. The forecast is that this supremacy will not decline in the future. Informal definition and enforcement is dramatically declining. Upper management is surprisingly absent as the main authority for either definition or enforcement. A dramatic shift is seen away from verbal transmission and toward primary reliance upon written policy, with a marked trend to on-line interactive access to written policy.

Introduction

"We have met the enemy - and he is us!" Pogo

The idea for this survey was stimulated when one of the authors suffered extreme curiosity about how policy[1] has been defined, promulgated, and enforced within the multi-user community, and how those practices might evolve in the future. The only source at hand to satisfy his questions was the large number of administrators[2] who read those Usenet newsgroups focussing on such issues.

He designed and posted a survey questionnaire, then built scripts to process the returned email responses into single-line records stored to a simple database[3]. A total of 293 responses were validated and tabulated.

Both authors then reviewed the data independently, merging their joint findings into this report.

Particular weight should be placed on the word *informal* in the title of this paper. No representation is made that this work fully qualifies as "real science": the sample size is small and self-selected, and the sample population cannot be verified to directly represent any specific population of multi-user - or even UNIX - installations owing to a variety of obvious biases.

In spite of such deficiencies, however, we believe this survey offers a very useful window on general practices and trends in policy administration, particularly for large-scale installations. As far as we are aware, it offers the first concrete data to be brought forth on this subject to-date.

Sample Population Demographics

Response Item	Resp.	%
My primary job location is in:		
Canada or U.S.	234	79.9%
Europe	40	13.7%
All Other	19	6.4%
My local organization is primarily:		
Educational	131	44.7%
Commercial	104	35.5%
All Other	58	19.8%
Multi-user facilities locally available:		
5 Years or Less	59	20.1%
More Than 5 Years	94	32.1%
More Than 10 Years	140	47.8%
I have total sysadmin experience of:		
5 Years or Less	161	54.9%
More Than 5 Years	104	35.5%
More Than 10 Years	28	9.6%
I have total user experience of:		
5 Years or Less	47	16.1%
More Than 5 Years	100	34.1%
More Than 10 Years	146	49.8%
My primary system admin duty is on:		
Un*x (any)	241	82.3%
VMS	24	8.2%
PC LAN	0	0.0%
Combo Of Above	22	7.5%
Other M/U	6	2.1%

Exhibit 1: Demographics

[1]For purposes of this research, "policy" is defined as all formal or informal rules of use of - or access to - multi-user computing resources.

[2]The terms "administrator", "system administrator", and "sysadmin" are here used interchangeably.

[3]This process is more fully described in the Appendix, which includes the text of the original Survey Form containing the specific language of the instructions, questions, and answers - often shortened for brevity in our exhibits.

Demographics

The sample population is described in the table shown in Exhibit 1.

The overwhelming majority (80%) of responses came from the U.S. and Canada. Educational institutions contributed the largest response (45%), followed by commercial enterprises of all types (36%). Most are UNIX sites, with a small number of VMS and "combo" sites. These serve an estimated total of nearly one hundred thousand users.

Respondents to the survey tend to be highly experienced users, and almost half report more than 5 years experience in system administration.

Note that respondents to the survey represent a very large number of users (see Exhibit 2).

We believe this survey best represents larger, older installations, and not smaller, newer sites. This would have particular impact on commercial representation, where such sites would more likely exist. Had these been fully represented, we suspect that the overall picture in the commercial world would have been even less rosy than we see here.

Total users on all systems administered:			
1 To 5	8	2.7%	20
6 To 25	41	14.0%	636
26 To 50	40	13.6%	1,520
51 To 100	49	16.7%	2,499
101 To 500	82	28.0%	24,641
501 To 1000	23	7.9%	17,262
More Than 1000	50	17.1%	50,000
Total estimated users			~97,000

Exhibit 2: Number of users administered

Practices & Trends

There are important similarities and differences between educational and commercial sites, and these two primary populations will receive the focus of

Practices & Trends - All Sites

Total Responses = 293

Response Item	Responses*	Past	Future	Trend
(A) Policies (written or not) include:				
Both guidelines and specifics	159/207	54.3%	70.7%	+16.4%
Specific Dos and Don'ts only	6/6	2.1%	2.1%	0.0%
Neither guidelines nor specifics	32/13	10.9%	4.4%	- 6.5%
General guidelines only	87/52	29.7%	17.8%	-11.9%
(B) Policies mainly defined by:				
User-committee decisions	11/33	3.8%	11.3%	+ 7.5%
Sysadmin	153/161	52.2%	55.0%	+ 2.8%
Upper management	20/27	6.8%	9.2%	+ 2.4%
User's immediate supervisor	5/3	1.7%	1.0%	- 0.7%
Informal user-practice	95/57	32.4%	19.5%	-12.9%
(C) Enforcement authority mainly from:				
Sysadmin	191/198	65.2%	67.6%	+ 2.4%
Upper management	25/30	8.5%	10.2%	+ 1.7%
User-committee decisions	8/10	2.7%	3.4%	+ 0.7%
User's supervisor	17/17	5.8%	5.8%	0.0%
Peer pressure of users	17/13	5.8%	4.4%	- 1.4%
No one	28/13	9.6%	4.4%	- 5.2%
(D) Policy mainly presented to users by:				
On-line interactive	84/152	28.7%	51.9%	+23.2%
Written - mgr, supr, sysadmin	44/52	15.0%	17.8%	+ 2.8%
Verbal - from other users	63/29	21.5%	9.9%	-11.6%
Verbal - mgr, supr, sysadmin	96/44	32.8%	15.0%	-17.8%

* First number is "past" (historical) sum, second is "future" sum.
NOTE: "New site" and "Don't know" responses excluded from listing.

Exhibit 3: Practices and Trends from all sites

our attention and commentary.

Reported practices and trends are summarized in Exhibits 3 through 5 for All Sites, Commercial only, and Educational only. These data are then summarized into comparative ''bar graphs'' for commercial and educational sites in Exhibits 7 through 9. All are cross-referenced throughout by alpha labels (placed in parentheses).

Policy Features(A)

Q: *Policies (written or not) did/will include what features?*

General policies alone are clearly no longer sufficient to satisfy the majority (70%) of sites, and in both commercial and educational institutions there is a surge toward addition of specific ''dos and don'ts'' to more fully implement (and more easily enforce) such general policy.

Reliance upon no policy at all is rapidly fading, and only a very few sites attempt to operate using specifics alone.

Commercial	Past	Future
Gen & Spec	47.1%	66.4%
Spec Only	1.9%	1.9%
General only	34.6%	19.2%
Neither	13.5%	3.9%

Educational	Past	Future
Gen & Spec	56.5%	72.5%
Spec only	3.1%	2.3%
General only	29.8%	19.1%
Neither	8.4%	5.3%

Exhibit 6: Policies (written or not)

While overall progress of these trends will be further advanced at educational sites, the percentage of change is forecast to be more pronounced at

Practices & Trends - Commercial

Total Responses = 104

Response Item	Responses*	Past	Future	Trend
(A) Policies (written or not) include:				
Both guidelines and specifics	49/69	47.1%	66.4%	+19.3%
Specific Dos and Don'ts only	2/2	1.9%	1.9%	0.0%
Neither guidelines nor specifics	14/4	13.5%	3.9%	- 9.6%
General guidelines only	36/20	34.6%	19.2%	-15.4%
(B) Policies mainly defined by:				
Sysadmin	53/61	51.0%	58.7%	+ 7.7%
Upper management	3/8	2.9%	7.7%	+ 4.8%
User-committee decisions	0/4	0.0%	3.9%	+ 3.9%
User's immediate supervisor	3/2	2.9%	1.9%	- 1.0%
Informal user-practice	42/25	40.4%	24.0%	-16.4%
(C) Enforcement authority mainly from:				
Upper management	6/10	5.8%	9.6%	+ 3.8%
User's supervisor	10/11	9.6%	10.6%	+ 1.0%
User-committee decisions	1/1	1.0%	1.0%	0.0%
Sysadmin	62/62	59.6%	59.6%	0.0%
Peer pressure of users	9/8	8.7%	7.7%	- 1.0%
No one	13/8	12.5%	7.7%	- 4.8%
(D) Policy mainly presented to users by:				
On-line interactive	23/43	22.1%	41.4%	+19.3%
Written - mgr, supr, sysadmin	7/14	6.7%	13.5%	+ 6.8%
Verbal - from other users	29/20	27.9%	19.2%	- 8.7%
Verbal - mgr, supr, sysadmin	43/20	41.4%	19.2%	-22.2%

* First number is ''past'' (historical) sum, second is ''future'' sum.
NOTE: "New site" and "Don't know" responses excluded from listing

Exhibit 4: Commercial Practices & Trends

commercial installations. See Exhibit 6.

Definition Authority (B)

Q: *Actual policy-definition authority* mainly *from where?*

Above all, the sysadmin now clearly dominates the entire field as a virtual dictator (hopefully viewed as benevolent), and the forecast by respondents is that this primacy will carry into the future.

Commercial sites appear to be extending even further this prevailing dominance of the sysadmin for authority in definition, with only minor movement toward reliance upon upper management or user-committees.

Educational sites show a somewhat higher (though still not large) reliance on upper management for policy definition. But they also are expanding almost singularly toward more user-committees, with neither the system administrator nor upper management showing noticeable future gains in this key area. (This visible trend may be a significant harbinger of the future, and we will focus more attention upon its possible interpretation hereafter).

Commercial	Past	Future
Sysadmin	51.0%	58.7%
Upper mgmt	2.9%	7.7%
User-comm	0.0%	3.9%
Supv	2.9%	1.9%
Informal	40.4%	24.0%

Educational	Past	Future
Sysadmin	54.2%	53.4%
Upper mgmt	9.9%	10.7%
User-comm.	7.6%	16.0%
Supervisor	0.8%	0.8%
Informal	24.4%	16.0%

Exhibit 7: Definition authority

Practices & Trends - Educational

Total Responses = 131

Response Item	Responses*	Past	Future	Trend
(A) Policies (written or not) include:				
Both guidelines and specifics	74/95	56.5%	72.5%	+16.0%
Specific Dos and Don'ts only	4/3	3.1%	2.3%	- 0.8%
Neither guidelines nor specifics	11/7	8.4%	5.3%	- 3.1%
General guidelines only	39/25	29.8%	19.1%	-10.7%
(B) Policies mainly defined by:				
User-committee decisions	10/21	7.6%	16.0%	+ 8.4%
Sysadmin	71/70	54.2%	53.4%	+ 0.8%
Upper management	13/14	9.9%	10.7%	+ 0.8%
User's immediate supervisor	1/1	0.8%	0.8%	0.0%
Informal user-practice	32/21	24.4%	16.0%	- 8.4%
(C) Enforcement authority mainly from:				
Sysadmin	93/98	71.0%	74.8%	+ 3.8%
User-committee decisions	4/6	3.1%	4.6%	+ 1.5%
Upper management	13/14	9.9%	10.7%	+ 0.8%
User's supervisor	4/4	3.1%	3.1%	0.0%
Peer pressure of users	6/3	4.6%	2.3%	- 2.3%
No one	9/2	6.9%	1.5%	- 5.4%
(D) Policy mainly presented to users by:				
On-line interactive	43/83	32.8%	63.4%	+30.6%
Written - mgr, supr, sysadmin	25/21	19.1%	16.0%	+ 3.1%
Verbal - from other users	22/7	16.8%	5.3%	-11.5%
Verbal - mgr, supr, sysadmin	40/15	30.5%	11.5%	-19.0%

* First number is "past" (historical) sum, second is "future" sum.
NOTE: "New site" and "Don't know" responses excluded from listing

Exhibit 5: Educational Practices & Trends

Regardless of which way power is migrating, both kinds of sites are noticeably abandoning the *laissez faire* notion of policy based on informal user-practice. Even so, some 16% of educational sites and 24% of commercial sites say they will remain with this approach in the future. See Exhibit 7.

Enforcement Authority (C)

Q: *Actual policy-enforcement authority* mainly *from where?*

Both educational and commercial sites are moving to vest enforcement power in a defined authority, rather than leaving it to user peer-pressure (or no one), so *laissez faire* benign neglect is declining in this category, also.

The sysadmin is supreme enforcement authority at well over half of all sites of both kinds, and the forecast is that this power will increase slightly at educational sites.

At commercial sites, no change is forecast for sysadmins, and the marginal gain goes mostly to top management (still an insignificant player). See Exhibit 8.

Commercial	Past	Future
Upper mgt	5.8%	9.6%
Supv	9.6%	10.6%
User-comm.	1.0%	1.0%
Sysadmin	59.6%	59.6%
Peer pressure	8.7%	7.7%
No one	12.5%	7.7%

Educational	Past	Future
Sysadmin	71.0%	74.8%
User-comm.	3.1%	4.6%
Upper mgmt	9.9%	10.7%
User's sup'vis	3.1%	3.1%
Peer pressure	4.6%	2.3%
No one	6.9%	1.5%

Exhibit 8: Enforcement Authority

Policy Presentation (D)

Q: *Policy information presented to users* mainly *by what means?*

Verbal instructions are clearly going to decline in favor of written policies of some form, and some of this shift will be to written policies provided in hard-copy only.

But the most numerically dramatic change will be toward on-line interactive presentation. At educational institutions this is forecast to be a whopping 30% increase, and almost 20% at commercial sites, thus catapulting on-line written presentation to become the primary format in both camps, a thorough reversal from historic reliance primarily on verbal transmission.

From data not shown here, we also note that the percentage of sites relying on written policy increases directly with size. Such increasing formality seems to be a natural companion of growth, so this finding conforms to our personal intuition based on professional experience. See Exhibit 9.

Commercial	Past	Future
On-line	22.1%	41.4%
Written	6.7%	13.5%
Verbal - users	27.9%	19.2%
Verbal - SA	41.4%	19.2%

Educational	Past	Future
On-line	32.8%	63.4%
Written	19.1%	16.0%
Verbal - users	16.8%	5.3%
Verbal - SA	30.5%	11.5%

Exhibit 9: Policy Presentation

Observations

It is most encouraging to see the increasing trend toward "putting it in writing" (of one form or another), and it is likewise reassuring to know that users will more often have convenient access to policy directly from their own terminals through interactive software – a change that should dramatically simplify administration and facilitate compliance.

As in certain aspects of programming or winemaking, some things work well, and some simply don't. Written policy is one of those things that works very well, indeed, if the aim is to create a system of real mutual accountability[4] for users, sysadmins, and managers and thus promote a predictable environment which reliably supports both user needs and the organizational mission.

Some may be surprised to discover that written policy (properly designed and presented) delivers profoundly greater value than was ever anticipated. It tends to inoculate the environment against a host of hidden stresses and dangers which formerly escaped conscious observation, or were simply dismissed as unalterable "givens".

Mission is the necessary antecedent of *policy*. Once mission is clear, then policy will more likely be:

- Easier to define and quicker to gain upper management endorsement.
- Briefer in verbiage, but more comprehensive in scope.
- More cohesive and logically consistent.

[4]For some, this may also be its most worrisome aspect, eliciting exclamations of concern about "excessive bureaucracy".

- More willingly supported by users, thus infrequently challenged.
- More-easily enforced if challenged, and less subject to the deadly disease of "politics"[5].

Ultimately, it is ordinary human behavior which most heavily bears upon the nature of the user environment, and the aim of policy is to regulate behaviors toward enhancing the quality of that environment for the collective benefit of all the parties having an interest. Policy is the blueprint for how this benefit is to be defined, and how achieved.

The effectiveness of local policy thus derives more from an appreciation of civics than of physics, and the primary considerations are those of governance: *policy is for people, not machines.*

The most revealing aspect of this survey, therefore, is the absence of upper management as primary authority for policy definition and enforcement, and how thoroughly the sysadmin single-handedly straddles both of these key functions.

One of the authors quotes a particular system administrator as having told him: "If I need to get top management involved, then I'm not doing my job right!" This anecdote may offer some insight to the survey findings, which signal a perilous disregard of organizational fundamentals, and strike to the very heart of "governance".

The current luxury of dictatorship by the sysadmin (where practiced) will prove both short-sighted and dangerous - not least to the sysadmin. To be performed successfully and gracefully, such role requires a degree of omnipotence, omnipresence, and omniscience which few mortals would feign. Absent these godlike attributes, either success or grace will likely suffer. Usually both.

While it is appropriate that both definition and enforcement be placed in the hands of formal authority, it is very doubtful that the system administrator should directly wield that scepter as his or her own.

Some educational institutions appear to be forming a vanguard to give definition authority to user-committees. This alternative may well be an attempt to relocate primary authority away from both of the dominant historical models of "benevolent dictatorship" and "benign neglect".

Even though well-intentioned, this is a misdirected effort. While we fully agree that there is an appropriate and useful function for user-committees, vesting them with *primary* authority for policy definition again assigns responsibility in the wrong direction.

There is one salient fact: upper management is the single authority charged with operational custodianship for all organizational assets, and it is inescapable that this extends to computing resources. Therefore, review and approval from this level is mandatory to assure that intended and actual usage of those resources conforms to the organizational mission which those resources are dedicated to support.

Users certainly have an important stake in the proper definition of policy, and it makes perfectly good sense to actively engage them to contribute to such development *via* a structured and representative body - i.e., a user-committee.

These two parties - upper management and the user-committee - might thus join powerfully, each ably championing those vested interests which it is best equipped by experience and knowledge to articulate. And each, likewise, would be compelled to more fully appreciate the valid concerns of the other, a precondition for consensus and compromise. The sysadmin, of course, remains a central contributor, particularly with regard to issues which are influenced by technical components.

But upper management, regardless of any unique features of the institution, simply cannot flinch from primary policy responsibility, and has the only the real authority to say "yea" or "nay" to ultimate policy definition or enforcement. Any governing process which omits this vital step is surely ticketed for eventual trouble.

If upper management is ill-equipped by knowledge or inclination to exercise this duty, then to hot-wire fancy schemes to obscure this fact simply abets the failure, compounding rather than addressing it.

This is not to say that upper management needs be the main source of policy proposals. Indeed, these might best be hammered out between representatives of the primary parties having an interest: upper management, users, and the system administrator.

This should be undertaken within the framework of an approved project, appropriately resourced and managed toward the goal of a sound "product" in the form of a mutually-agreed written policy that obeys fundamentals, garners wide-spread support, and is underwritten by upper management (at the highest level obtainable).

This end product deserves to be properly sponsored, defined, developed, documented, tested, and timely delivered to the customer. The only real difference here is that now the "customer" is *us*, and we must live with whatever success or failure is generated from our strategy for accomplishing these tasks.

[5]E.g., "power users" can become formidable dragons when the sysadmin solely creates policy on his or her own authority, only to discover in sorrow that sufficient clout is lacking to enforce it uniformly when these users can successfully assert a claim that they are "more equal" than others.

Pogo's Maxim should thus caution us to avoid flawed notions of governance, old or new, which fail to address immutable realities.

Author Information

Bud Hovell, a principal of MTEK International, Inc., has been a technical management consultant for 15 years and is a certified PMP and CPIM. He has furnished previous articles for *UNIX REVIEW* on computer resources policy, and is primary developer of The Policy Package, a freely-distributed kit of interactive UNIX scripts for implementing simple on-line policy administration. Bud can be reached by email at *bud@mtek.com*, or by telephone at 1-503-636-3000.

Bjorn Satdeva is the President of /sys/admin, inc., a consulting firm which specializes in Large Installation System Administration. Bjorn is a member of the IEEE POSIX 1003.7 System Administration Standardization Committee. Bjorn is also President and co-founder of Bay-LISA, a San Francisco Bay Area user's group for system administrators of large sites, and Senior Editor for ROOT, the UNIX System Administration Magazine. Bjorn can be reached by email at *bjorn@sysadmin.com*, or by telephone at 1-408-241-3111

Appendix: Survey Methodology

In basic concept, the survey questionnaire follows the familiar multiple-choice format. There are a few new wrinkles, however.

|AD|	Europe, except Soviet Union
|BA|	Commercial - Hardware (...) manufacture
|CG|	More than 5 years
|DG|	More than 5 years
|EH|	More than 10 years
|FG|	More than 1000 users
|GA|	Un*x [any flavor]
|HB|	Specific "Dos & Don'ts"
|ID|	Sysadmin or system manager decisions
|JE|	Sysadmin or system manager
|KB|	Verbal info from (...) sysadmin
|LB|	Probably about once a year
|MA|	Virtually never
|NC|	Both of the above
|OD|	Sysadmin or system manager decisions
|PE|	Sysadmin or system manager
|QD|	On-line interactive (...) policies
|Z2|	Usenet Survey

Exhibit 10: Abridged Survey Response Example

An important design criterion - ease of use - was achieved by a simple "elimination" method. That is, each available answer is placed on its own appropriately-coded line, so undesired responses to each question need only be deleted on a line-by-line basis using a text editor.

This leaves a short document (see Exhibit 10) of surviving desired responses for return by email to the survey address. The convenience afforded for the respondent may well encourage higher return-rates than methods which simply emulate the mechanics of paper-oriented techniques.

```
SYSMGR@MTEK.COM|05/03/91|AD|BA|
    ...|NC|OD|PE|QD|Z2|
```

Exhibit 11: Abridged data record

Exhibit 11 shows how a processed data record for one respondent might appear in the final database. Pipe-delimited (|) fields contain the respondent's email-id, date received, and thereafter the full set of responses chosen by him or her in reply to the survey questions.

Implementation

The survey questionnaire was distributed by cross-posting to three Usenet newsgroups solely dedicated to system administration subjects:

- comp.unix.admin
- news.admin
- vmsnet.admin

Except for an omission in the original instructions, which therefore had to be hastily revised and re-issued, this methodology otherwise worked extremely well, and subsequent conduct of the survey proceeded without difficulty.

Errata

Some readers may have been made curious by the absence of comment about the two questions on the survey asking about the frequency at which management and users asked questions regarding policy.

It was our opinion that the questions produced doubtful results, eliciting responses tending to display a pattern of "central tendency" (taking the middle answer) which usually occurs when the respondent must select among arbitrary choices about which he or she can make no positive discrimination. The fault, if any, lies with the questions themselves, not with the effort of the respondents.

The surveying author must frankly admit that he had some doubt about the merit of these two questions when he finally chose to include them. On further reflection, he might have decided otherwise. Hindsight *is* "20/20".

Since this survey may be further extended to a broader sample, however, we will examine again later whether this pattern holds up or not. For now, we believe the data require us to reserve judgement.

Survey Form

========== 'POLICY' SURVEY ==========

PURPOSE
This survey asks system administrators (including news administrators) for brief feedback about local "policy" in their multi-user computing environments, UNIX or other.

For this research, 'policy' is defined as all formal or informal rules of use of - or access to - such computing resources.

PRIVACY OF INFORMATION PROVIDED
Your name and/or email address will *not* be released to - nor your individual responses be shared with - any person, agency, or organization which is not directly engaged in performing this research. You will not be solicited by anyone nor be otherwise annoyed as a result of your decision to respond or not respond.

INSTRUCTIONS
Those persons ONLY should respond who have some current responsibility for system administration, or who supervise others who do. Such duty need not be one's sole (or even primary) duty.

Under *each* header below, select your *one* best choice by *deleting* all others under that header (and the header itself), then proceed on to the next header. This is a forced-choice, multiple-choice, complete-the-statement format. Just like in school: even if you think the question is perfectly lousy, pick just one anyway. Please.

[If you wish to supply comments, they are *most* welcome - but please send them back by separate mail with "Subject: comments" (or suchlike) so we can pick them out from survey-responses, which get automated (sorta) processing on this end.]

You should be able to work these in order from top to bottom, deleting as you go. When you are done, you should have remaining a total of exactly 18 lines of actual responses. Other lines of text will be ignored.

FUNNY STUFF
It shouldn't matter if your mailer inserts ">" or other customary left-margin marks when you reply - you send it, we'll parse it :-). If you have your own dot-sigs or other stuff containing pipe-marks scattered about, however, it just means someone here may have to hand edit them out. You *could* save us that minor misery by doing this yourself. :-)

========== BEGIN SURVEY ==========

You can start by deleting everything above this line.

(Basic Info)

My *primary* job location is in:
==========
|AA| Africa
|AB| Australia
|AC| Canada
|AD| Europe, except Soviet Union
|AE| Far East, except Japan and Soviet Union
|AF| Indian Subcontinent
|AG| Japan
|AH| Mexico and Central America
|AI| South America
|AJ| Soviet Union
|AK| United States
|AZ| Other

```
My local organization is *primarily*:
======================================
|BA| Commercial - Hardware or software manufacture
|BB| Commercial - Hardware or software sales or service
|BC| Commercial - Other manufacture
|BD| Commercial - Other sales or service
|BE| Educational (university or other)
|BF| Governmental, except military
|BG| Health-care
|BH| Military
|BI| Religious or fraternal
|BJ| Research, except educational
|BZ| Other

Multi-user facilities of some kind have been locally available:
================================================================
|CA| Fewer than six months
|CB| More than six months
|CC| More than  1 year
|CD| More than  2 years
|CE| More than  3 years
|CF| More than  4 years
|CG| More than  5 years
|CH| More than 10 years

I have total system administration experience of:
=================================================
|DA| Fewer than six months
|DB| More than six months
|DC| More than  1 year
|DD| More than  2 years
|DE| More than  3 years
|DF| More than  4 years
|DG| More than  5 years
|DH| More than 10 years

I have total experience as a user (including sysadmin) of:
==========================================================
|EA| Fewer than six months
|EB| More than six months
|EC| More than  1 year
|ED| More than  2 years
|EE| More than  3 years
|EF| More than  4 years
|EG| More than  5 years
|EH| More than 10 years

Total users on all systems I currently administer:
==================================================
|FA| 1-5 users
|FB| 6-25 users
|FC| 26-50 users
|FD| 51-100 users
|FE| 101-500 users
|FF| More than 500 users
|FG| More than 1000 users

My primary administration activities are on systems which are:
==============================================================
|GA| Un*x (any flavor)
|GB| VMS
|GC| PC LAN
|GD| Combination - 2 or more of the above
|GZ| Other multi-user
```

```
(Historical Practice)

Historically, our policies (written or not) have included:
==========================================================
|HA| General guidelines
|HB| Specific "Dos & Don'ts"
|HC| Both of the above
|HD| None of the above
|HE| Don't know
|HZ| New site - no historical practice

Historically, actual policy has been *mainly* defined by:
=========================================================
|IA| Informal day-to-day user practices
|IB| User-committee decisions (or similar formal means)
|IC| Directions of user's immediate supervisor
|ID| Sysadmin or system manager decisions
|IE| Upper management decisions
|IF| Don't know
|IZ| New site - no historical practice

Historically, actual policy-enforcement authority came *mainly* from:
=====================================================================
|JA| No one - each user did what he needed to do
|JB| Users, through active peer pressure
|JC| User-committee decisions (or similar formal means)
|JD| User's immediate supervisor or manager
|JE| Sysadmin or system manager
|JF| Upper management
|JG| Don't know
|JZ| New site - no historical practice

Historically, policy information was presented to users *mainly* by:
====================================================================
|KA| Verbal info, primarily from other users
|KB| Verbal info from manager, supervisor, or sysadmin
|KC| On-paper-only written info from manager, supervisor, or sysadmin
|KD| On-line interactive displayed (as well as on-paper) written policies
|KE| Don't know
|KZ| New site - no historical practice

(Current Interest)

Recently, users raise policy questions:
=======================================
|LA| Virtually never
|LB| Probably about once a year
|LC| Probably about once a month
|LD| Probably about once a week
|LE| More frequently

Recently, middle and upper managers raise policy questions:
===========================================================
|MA| Virtually never
|MB| Probably about once a year
|MC| Probably about once a month
|MD| Probably about once a week
|ME| More frequently

(Future Expectations)

Future policies (written or not) will include:
==============================================
|NA| General guidelines
|NB| Specific "Dos & Don'ts"
|NC| Both of the above
|ND| None of the above
|NE| Don't know
```

```
Future actual policy will be *mainly* defined by:
=============================================
|OA  Informal day-to-day user practices
|OB  User-committee decisions (or similar formal means)
|OC  Directions of user's immediate supervisor
|OD  Sysadmin or system manager decisions
|OE  Upper management decisions
|OF  Don't know

Future actual policy-enforcement authority will *mainly* come from:
====================================================================
|PA  No one - each user will do what he needs to do
|PB  Users, through active peer pressure
|PC  User-committee decisions (or similar formal means)
|PD  User's immediate supervisor or manager
|PE  Sysadmin or system manager
|PF  Upper management
|PG  Don't know

Future policy information will be presented to users *mainly* by:
=================================================================
|QA| Verbal info, primarily from other users
|QB| Verbal info from manager, supervisor, or sysadmin
|QC| On-paper-only written info from manager, supervisor, or sysadmin
|QD| On-line interactive displayed (as well as on-paper) written policies
|QE| Don't know

Please include this next line:
==============================
|Z2| Usenet Survey

*************************************************************************
   Now delete all lines that are not responses to questions, and please
       email to "survey@mtek.com" or "tektronix!bucket!mtek!survey"
*************************************************************************
```

Enhancing Your Apparent Psychic Abilities Through Software

Elizabeth D. Zwicky - SRI International

ABSTRACT

This short short describes a way to monitor other users' screens.

I work in California, where long, flat buildings are popular, due to the problems with tall, skinny buildings falling over in earthquakes. The building I'm in is a very long flat building, and our user services desk is all the way at one end of it. After I had spent enough time trudging to the far end to fix mystery problems that became clear as soon as I saw the screen, I remembered a program called ''spy'', which Tom Fine had created some years ago at Ohio State as a proof that Sun's letting random people mmap the frame buffer when other people were using it was a security problem. I revived it, fixed some of its worst problems, and put it into use.

''spy'' is actually two programs. On the workstation you are sitting at, you run the server, which is called ''spyin''. It mmaps the frame buffer, binds to a socket, and waits. On the machine you want to spy on, you run the client, ''spyout'', telling it what machine the server is on and how many screen copies you want from it. It connects to the server and sends its framebuffer type; the server closes the connection if the client framebuffer is not the same type as the server. The client then reads the framebuffer directly into the socket; the server reads the socket directly into the framebuffer. When the client has read the entire frambuffer the specified number of times, it exits.

The result is that your screen becomes a duplicate of the remote screen. Neither machine needs to be running a window system at all. If the remote screen has been locked, you see the lock; on the other hand, screenblank works by turning off the display, not by changing the bits in the framebuffer, so you actually see what's underneath. Your window system will fight with spy for control of your framebuffer (you can actually use this for load estimation if your clock has a second hand, since on a normally loaded system you get about a remote screen a second).

The drawbacks of this system are mostly obvious. It makes for really obnoxious screen-junk on the local screen, it isn't robust, it's specific to machines that let you mmap the framebuffer and if incautiously used it makes the security hole it was originally designed to point out all too obvious. (Yes, Sun's default configuration does still let anybody access the framebuffer, just like it lets people use the audio on your SparcStation to drive you nuts.) Furthermore, on a Sparc GX running 4.1, mmaping the framebuffer not only doesn't work, it also causes the machine to turn off video and ethernet. (The version I run simply refuses to run the client on colour machines.) A cleverer implementation could avoid most of these problems.

Even this crude implementation is a fantastically useful tool in our environment. It's amazing how many things are easier to diagnose if you can see them - users just aren't very good at describing them. What really puzzles me, however, is that only about half the users ever catch on to the fact that you are looking at their screen. Half of them are not in the least bothered when they call you up and you correct typos as they're typing, or tell them where to look on their screen. (We do tell them - aside from ethical considerations, it's not in our best interests to encourage them to believe that we really are psychic. It just makes them sulkier when we can't guess what they're thinking.)

Author Information

Elizabeth Zwicky is an insufficiently psychic system administrator for the Information, Telecommunications, and Automation Division at SRI International. Reach her via U.S. Mail at SRI International; 333 Ravenswood Avenue; Menlo Park, CA 94025. Reach her via electronic mail at zwicky@erg.sri.com.

Engineering a Commercial Backup Program

Jeff Polk & Rob Kolstad - SunSoft, Inc.

ABSTRACT

Many customers wonder why vendors do not support them with quick and accurate responses to their requests for new software. It is now true that, for whatever reason, the computer industry has reached a point where customers can pose problems faster than any vendor or third party can solve them. While this is partly a function of galloping hardware improvements (and sometimes of escalating promises made by non-engineering groups), it is primarily a function of the complexity of engineering a piece of software that runs correctly on thousands of machines.

The productization of any particular software requires analysis of many design issues and choosing trade-offs between functionality, reliability, cost of implementation, and time to market in addition to the little discussed notions of testing.

This paper describes the issues behind the engineering of a commercial backup program. These include not only implementing the code itself, but also analyzing and engineering performance, installation, testing, and release.

Introduction

Customer's expectations of computer vendors are fueled by media hype and an eager sales force. Vaporware is too often the rule rather than the exception.

Customers then wonder why their expectations are not being met. This paper uses a case study to demonstrate the theory and practice behind releasing a backup program.

The paper discusses the interactions among the various groups that cooperate to create a product, the creation and execution of various schedules, and the reality behind the engineering and back end processes.

The Case Study

The Backup Copilot product recently announced by Sun Microsystems makes a good example for study. The system consists of seven major functional components:

- Dump
- Restore
- Recover
- Database
- Operator interface
- Sequencer
- Disaster recovery

along with documentation, market collateral, and test suites.

The improved version of *dump*(8) accepts many new options and performs on-line dumps [1]. It communicates with the operator monitor system and the updates an on-line database of dump indexes. This new version of dump also recognizes EOT instead of performing calculations based on tape density and length. The changes for dump include thousands of lines of new code and extensive changes to existing code for a total of 8,420 code lines - of which 6,000 were new.

The new version of *restore*(8) [2] includes support for the special interface used by the new recover program and support for the new *f* option syntax to support multiple input devices and/or device sequences. The new version of restore includes 7,687 lines of code, of which only 1,300 are new.

The new *recover*(8) utility [2] is an interactive program that uses the database system to present a view of the filesystem tree as it appears in dump images. Users can `cd`, `ls`, obtain *stat*(2) information, and learn what versions of files or directories exist on backup media. They can request recovery of files, directories, or complete file systems. The recover utility will schedule tape mounts and invoke the enhanced restore program to recover the requested data. *recover*(8) consists of 8,076 lines of new code.

The database system [2] maintains a database of files and directories that exist on dump media. The database is updated by *dump* and is used by *recover*. It consists of a database daemon (8,438 lines of code) and a database maintenance utility (3,214 lines of code). The daemon manages all interaction with the database files and performs crash recovery on startup. The maintenance utility provides a means of quiescing the database, adding or deleting dump index information manually, listing tape/dump information, and recovering database files from some kinds of failures.

The operator interface system [3] consists of a daemon and an interface program. The daemon receives messages from various utilities and forwards those messages to other operator daemons and operator monitors. Responses received from operator monitors are forwarded back to the original requesting application. The operator daemon performs contention resolution on responses so that only one response is ever received by the original application. The operator monitor is a curses based program that displays messages in a scrolling buffer and allows the user to respond to queries from anywhere on the network. The monitor comprises 2,890 lines of code; the daemon 2,060 lines.

The execution system [4] uses a configuration file and a tape library file with tape label and expiration information to sequence dumps for a site. It can execute dumps on the local machine and/or remote machines using local and/or remote tape drives. It utilizes many of the new features of **dump** and ensures that the dump database is updated appropriately. The execution system consists of a dump execution program, two configuration programs, a tape library maintenance program, and miscellaneous programs and scripts to assist in disaster recovery. All told, the execution system is comprised of approximately 9,124 lines of C code and shell scripts.

The disaster recovery system ensures that files and filesystems can be restored even in the case of complete loss of disk storage. Recovery code is pervasive through some of the other utilities and includes 700 lines of shell scripts.

While the user level code comprises the vast majority of the project, the necessary kernel support adds significant requirements to the later phases of the approval and testing processes. The kernel modifications required for Backup Copilot amounted to approximately 2,500 lines of code and 420 lines of additional user code to access the new kernel functionality.

The total new code for the project was over 44,900 lines (since some old code lines were modified).

Requirements

Several sets of requirements constrain a product. The marketing group and the engineering group both generate requirements. Negotiations then ensue in order to reach consensus. In the case of Backup Copilot, the marketing requirements were extensive. A few of them are shown in Figure 1. Hollow bullets indicate those requirements which ultimately were not completely met by the product.

Of the requirements, some were cosmetic or user interface issues; some were new functionality; some were "overriding concerns". The most important ones were:

- Absolutely reliability: Integrity of user data is of primary importance. Any program responsible for protection and/or archival storage of that data must be as reliable as possible. Few utilities are more responsible for user data than the *dump* and *restore* programs. Users must be able to trust that data on their dump tapes is valid and correct so that they may recover from incidental losses or catastrophe.
- Ability to recover from all sorts of catastrophes. Since backups are most often made for the explicit purpose of mitigating disaster, a viable commercial backup program must provide a simple and straightforward way of recovering lost files and directories individually or as complete file systems.
- Robustness in light of various component failures. Any backup product that was not itself robust enough to recover from any common failures or could not at least continue to provide the basic functionality required of it in light of component failures would make a poor choice for safeguarding valuable data.
- Concinnity. All utilities on the system should interact in reasonable ways, and the old methods for doing dumps and restores were very unfriendly in this regard. The new backup system attempts to remedy many of the poor behaviors of the old way and leverage existing system facilities where possible accomplish this goal.
- Backward compatibility. Backward

● Unattended operation	● Flexible scheduling	● Increased performance
● Per file dumping	● Easy migration for scripts	● Absolute data integrity
● Multiple f/s on one tape	● Audit trail/log	○ Menu driven parameter setting
● Multiple tapes for one f/s	● Operator messages	○ Full GUIs for everything
● Sequences of output devices	● Online operation	○ Operator error detection
○ IBM tape labels	● Tape labels	● Easy-to-use operator interface
● On line index	○ ANSI tape labels	● End-user restore requests
? Meets all standards	○ Automatic file migration	● Tape library management
● Access security for database	● Ability to backup 512 spindles	○ Virtual tape support
● Access security for devices	○ Process mount/umount req's	○ Calculate time/media for backup

Figure 1: Some initial marketing requirements (○ → not met)

compatibility with previously written dump media is desirable for several reasons. In most cases, the new backup scheme will replace existing schemes, so sites that have been using the standard dump facility have a significant investment in data already on backup media in the old format. Also, if any changes were made to the media format, they would have to be supported for an extended period of time even if the new format was relatively short lived. By retaining the original media format, not only can the new dump scheme deal with dumps made with the old utility, the old restore utility can be used to restore files from media created with the new version of dump.

- Easy installation. To be successful, any software product must be easily installable and configurable in light of the many different machine and network types available today. The backup product epitomizes many of the problems encountered in trying to engineer software that is easy to use, yet is applicable to greatly varying requirements.
  Installation is relatively simple for individual machines, but large networks of machines exhibit problems since kernels must be installed on each machine while user code often needs to be installed only on servers (since many client machines often share user level executables). The installation problem is further exacerbated by the great variance in skill of administrators and the general lack of regard for documentation.
- Easy Configuration. Configuration is one of the most difficult problems faced by the engineering team. Customers demand that the system be completely configurable. They want complete control over all aspects of the system. They also want defaults that do exactly what they want and they want everything to be trivial and intuitive to change. Given infinite flexibility, it is impossible to make it trivial to understand; likewise, if you make the system trivial to configure and understand, you remove flexibility that some sites may actually need in order to use the product.
- Graphic User Interfaces. (This requirement was not met.) Another argument often surrounding ease-of-use issues concerns graphical user interfaces (GUIs). The decision on whether or not to implement an interface as a GUI is not a simple one when time to market is at issue. The engineers must trade off information from marketing and upper management that "you must have a GUI for it to be easy to use" against implementation deadlines, normal customer use of the product, and intended product use. In general, a GUI takes longer to design, implement and debug. GUIs also have the disadvantage of requiring bit-mapped displays and the use of a window system, so even in the presence of GUIs, a standard command line and/or curses interface is still often required.
- High performance for **dump**. Performance is also a critical design issue in a product of this type [5]. Present users expect a certain level of performance (from their experiences with the original *dump* and *restore* programs) and demand higher performance to support their ever-larger disk farms. The **dump** program is often one of a site's highest consumers of both disk accesses and CPU time. Since one key component of the new dump system is the enhanced version of **dump**, it was resolved that the performance should be able to exploit the speed of the new super-high density peripherals (e.g., 5GB Exabyte 8500 with a 500KB/second transfer rate).
- High performance for **restore**. The **restore** program, while not a frequent problem, has long been a problem for administrators due to its lack of speed for large restores (due to the the long time required to create a new file in the UFS file system). These long file create times are due to the synchronous operations the kernel performs to create a file and maintain a mostly consistent on-disk filesystem. The product team believed that administrators restoring a file system from scratch would gladly trade a 4-6x performance improvement on full restores for the necessity of starting completely over in the rare event of a machine crash in the middle of the restore.
  The other performance considerations in this product concerned the speed of the network transfers of data to the database and the speed of database update itself. Much of the database design was influenced by these considerations.

Before Engineering Begins

The design and implementation of a product is only a small part of the product engineering cycle; the path a product takes from inception to customer shipment is long and fraught with peril and frustration.

When the initial call for a product is made, all sorts of groups get involved to determine whether this particular call is one deserving attention at all. The product marketing organization does an analysis to decide this question and often provides many of the goals for a product. Marketing requirements are handed off to an engineering team whose job is to propose a solution to the problem posed by those requirements. The team often iterates on the requirements with marketing representatives in an

effort to maximize functionality yet specify a product that can be built in a reasonable time.

But the solution proposed by engineering is not straightforward, either. With the high costs of engineering a project today, the engineering organizations are under extreme pressure to leverage existing products in the marketplace wherever possible. Before coming up with an engineering proposal to create the project in-house, the engineering team must look to the marketplace for all similar products and see if any of those products currently meet the requirements (or could be made to meet the requirements).

It is often necessary at this stage to meet with these third parties and request proposals from them for custom products or for joint development of custom products. At this point all the corporate deal makers and management also get involved - this can be an extremely time consuming process. While some engineers are evaluating third party solutions, ideally another set is implementing a prototype of the key functionality of the product so that the accurate in-house engineering costs can be assessed.

If the engineers and management have decided that no third parties can supply the needed product for a reasonable cost (cost includes both financial and time-to-market elements), then the rest of the design/implementation process continues. At about this point, the engineering schedule is created and the true design and implementation phase begins.

Engineering

Theory

The backup project was begun just before June 1, 1990. The phases discussed earlier overlapped for about three months. The schedule was continually revised for months through the summer. By August, the schedule had firmed up (completion dates listed):

9/ 2/90	Functionality Overview
9/13/90	Schedule
10/25/90	Design complete
10/30/90	Arch. Comm. Approval
11/15/90	Database s/w complete
11/29/90	Dump program complete
12/14/90	Monitor complete
1/ 4/91	Configuration (sequencing) complete
1/ 4/91	Restore/recover complete
1/11/91	Installation paradigm complete
2/ 1/91	Alpha qualification complete
2/ 4/91	Alpha test begins

Note that actual implementation did not begin until October 28 - almost five months after the project had begun.

Engineering design has everything to do with writing documents and garnering approval for them. The engineering team wrote five documents (dump, operator monitor, sequencer, recover, and the database). These documents were completed between the middle of October and the middle of November (though, owing to the changed schedule, the sequencer document was never completed in design-document form).

The design and implementation phases take into account all the design criteria mentioned so far, along with the marketing requirements and input from a new player: the architecture committees. In large companies with many products in parallel development, it is necessary for some group to oversee changes to the software system that could possibly affect other products - and ensure that those changes don't conflict. Each product design is reviewed by the appropriate committee and any questions or concerns they raise must be addressed in the design before any product can be released. While obviously beneficial, architecture committee review and any redesign or change required can add to the completion time of the project. The architecture committee reviewed the backup project in January of 1991.

Concurrent with product design and development, the test development group plans how to test the functionality of the finished product. The test development group for the Backup Copilot product produced significant levels of code.

Experience

The initial engineering development team comprised two individuals: one for dump, the operator monitor, and sequencing. The other was responsible for restore, recover, and the database. The test organization devoted a single individual to test development. A manager was to coordinate the efforts while the writing team wrote the requisite documentation.

By the time Alpha qualification begins, all functionality must be in the product and only a few bugs might remain. The documentation may not be finished but all code is ready to go for internal production.

As it turns out, the schedule was more than ambitious. Little milestones turned out to be greater hurdles than anticipated. By December, it was clear that the schedule was unmeetable.

Ultimately, the original schedule (of about 15 weeks) was lengthened by seven weeks. An additional engineer was added to the team to implement the sequencer.

Many lessons were learned about project scheduling, particularly around the Christmas holidays.

Testing

Quality assurance requires many different types of tests. The test development group creates a large suite of tests that includes several major test

regimes:

- Regression testing: to ensure the product behaves in a backward compatible manner
- Conformance testing: to ensure the product meets its own functional specification
- Deviance testing: to ensure the product performs in a reasonable manner in light of failures or unexpected inputs
- Platform testing: to ensure the product performs on many different machines and configurations
- Cross-compatibility testing: to ensure the product does not interoperate in an undesired way with other supported products

The cross product of all of these tests is often of completely unmanageable size, so key points are chosen for testing. If any problems are discovered in testing, engineering fixes the problem and everything starts again.

The test group created 26,700 lines of code (well over half of it in shell scripts):

```
   700 Environment setup
 3,700 Test driver
 1,400 old dump regression
 4,500 new dump conformance
   400 of error injection (deviance)
 2,000 of expect scripts
 6,000 to test lockfs
 1,200 media tests (remote tape, EOT)
 1,400 recover
 1,800 restore
 3,600 miscellaneous utilities
```

The tests consumed thousands of hours of CPU time in their effort to verify the proper behavior of *dump* and *restore*. One cycle through the test suite in 'long mode' required about 36 real time hours. The group logged over 3,500 hours of testing before FCS.

The Back-end

As the implementation phase of a project draws to a close, one might expect that the product would soon be on its way to customers. Unfortunately (or fortunately, depending on your point of view), this is not the case. When implementation is mostly complete (implementation is never complete), the product enters the 'back end'. The back end refers to the official testing, manufacturing, and release procedures.

Customers who purchase products have very high expectations of their quality. Imagine if Lotus-brand spreadsheets occasionally miscalculated a column total. Buildings could fall; airplanes might tumble from the sky; entire third world economies might collapse. Likewise, Backup Copilot.

To avert such catastrophes, each product goes through a series of internal tests by the engineering organization, a series of tests by the company-wide product testing organization, and a period of use by actual customers for each of three releases: alpha, beta, and FCS (First Customer Shipment). The back end schedule has several components:

Alpha:

```
3/18/91 Freeze; RMTC tests begin
3/29/91 Mt. View. SQA Handoff
3/29/91 Alpha Begins
```

Beta:

```
3/20/91 Sites Identified
4/24/91 Freeze
4/24/91 RMTC SQE
5/ 8/91 RE & Doc Handoff
5/10/91 SQA Handoff
5/16/91 TOI
5/23/91 Pre-Master Burn/SQA
5/24/91 1st Article
5/30/91 1st Article SQA
5/31/91 CCB Signoff/RTF due
6/ 3/91 CD Production
6/ 5/91 Kit (S/W Manu.)
6/11/91 Beta Begins
```

FCS:

```
7/24/91 Freeze
7/24/91 Internal SQE
8/ 7/91 RE & Doc Handoff
8/ 9/91 SQA Handoff
8/30/91 Pre-Master Burn/SQA
8/31/91 1st Article
9/ 5/91 1st Article SQA
9/ 6/91 CCB Signoff/RTF due
9/ 9/91 CD Production
9/12/91 Kit (S/W Manu.)
9/17/91 for CSS
9/24/91 SHIP
```

As you can see from the schedule, this is not a trivial process in terms of resources or time. Sun runs three phases in its back end: Alpha test, Beta test, and FCS. Each of these phases must include the time it takes for release engineering (a separate organization from product engineering) to build (and possibly fabricate) release media and documentation. Testing then takes over and runs all the tests. All of these schedules necessarily span holidays and coordinate with other work-related demands; all of these considerations must be accounted for in the schedules.

Alpha Testing, Theory

For Alpha testing, a few errors are allowed to be in the product when it is shipped to those customers who will test it in production. Sun's alpha test ships the software only to sites within Sun's large internal network (20,000 machines).

Before the software is sent to the alpha sites, it is qualified by the testing group that works with the product group. When they are satisfied that they can assess its quality - and when the quality is good

enough – the software is sent to the alpha test sites and, along with the tests written by the test group, to the Mt. View testing organization. They test it in parallel with the alpha sites so that when it is "really ready to go", they have already encountered the special pitfalls or problems associated with the product.

Alpha Testing, Experience

Despite the best of intentions, the product required new functionality after Alpha had begun. Disaster recovery required more data to succeed – and ultimately would require an additional shell script.

Getting sites to install a new backup program turns out to be dramatically more challenging than one might initially guess. It seems that they are quite protective of their data!

Alpha testing revealed that the installation and configuration process was far too complex for the average system administrator. The configurability of the sequencer constituted a hurdle that few administrators were willing to jump as the first thing to execute after installation.

The configuration system was augmented with a new, easier configurator. Some minor bugs were found in Alpha testing, but not as many as might be desired.

Beta Testing, Theory

Beta testing puts the product into customers' hands and also begins the ramp-up throughout the company so that the entire corporation is familiar with the product. Beta testing brings in customer service, marketing (particularly training aspects), technical support engineers (in the field), and manufacturing.

By the time a product is ready for Beta testing, it should be thoroughly wrung out and have few errors (though some might remain). Alpha feedback should have improved the product. It should be almost of production quality. The test group may have developed additional tests by the time Beta qualification begins. The documentation should be pretty

At least five sites outside the company are chosen as Beta test sites. Sun's criteria for releasing a product includes the ability to cite at least five happy customers by the time the product is announced. Many times, more than five customers are chosen for Beta testing – the release criteria then grows in complication as various formulae are applied to the 'happy customer' index. They are chosen for their configurations, expertise, and tolerance of potential errors. They are the 'early adaptors'.

In addition to readying software for the customers, training is given to customer service. Hundreds of overhead slides and a six-hour presentation prepared the customer service people for the calls they might receive about the product.

Once the software is again frozen then tested locally, it is handed off to the Mt. View test people who test it on many configurations before blessing it. The software makes its way to the CD manufacturer where a small number of them are burned and shipped to Mt. View. They are verified against the master binaries and the 'go' signal is given to the manufacturer for a production run (small in the case of Beta) of the disks. Interestingly enough, the turnaround time through the CD manufacturer for CD software is a constant whether one needs 1 disc, 10 discs, or 10,000 discs!

Beta Testing, Experience

Changes both small and medium in scope continued right up to the cutoff for software testing. Most of these changes resulted from internal production testing, which had started well before the first freeze.

As new (quite unexpected) situations are encountered, new actions are taken to resolve them. Complexity continued to be reduced as configurability was maintained.

Beta sites were more willing to jump into testing the product, but illnesses, pregnancies, and vacations postponed many installations. The feedback loop (back through the customer service organization) was not utilized by the customers as much as one would think.

Final Engineering

In parallel with the alpha and beta test, the product engineering team runs performance evaluation tests to ensure that it meets any performance goals and to eliminate any implementation blunders that could cause poor performance.

Of course, bugs are eliminated as they arise. Tests are designed to re-expose them, should they surface anew.

Our bug tracking system recorded 393 bugs and requests-for-enhancement (RFEs) filed against the backup product. All 393 were resolved (though some RFEs were denied) when it came time for FCS freeze.

FCS, Theory

FCS is, of course, the big goal. Interestingly enough, it takes two months from FCS freeze before the first CD ships to a customer.

Between the freeze and the shipment, many parties must agree on procedures and execute:

- The marketing group must have a product

name chosen well in advance of the documentation handoff (8/7/91 for this product). The documentation must be handed-off on schedule so that it can be reproduced in vast quantities for customer distribution.

- The marketing group must have all kinds of support documents and collateral ready for distribution by the announcement date. This means they not only must design the collateral but also ensure that it is distributed and ready to go.
- The documentation group must not only have the man pages and user guide ready, they must supply the art work for the CDRom.
- Manufacturing must be ready with part numbers and a manufacturing slot that meshes the schedule between CD production and FCS.
- Customer service must be sure that they are ready to field calls about the product.

FCS freeze is an exciting time – the software is soon to be a Product that can produce revenue for the company.

FCS, Experience

It's hard to believe that a product name can be so difficult to conjure. Not only was it a problem to get the product name in the documentation on schedule, but there was some call to change it at a very late date. So far, the change has not been instituted. In a similar vein, whether to make a small documentation change became an intense debate which was eventually resolved but not before wailing and gnashing of teeth.

The company (Sun Microsystems, Inc.) partitioned itself into three entities, thus complicating interfaces and confusing the actual deliverables of the product itself. It is expected that this will not be a common occurrence.

Finally, a bug in the underlying kernel code was discovered on August 16 (some testing is going on in Mt. View, but it was not too late to push back the release). As this is being written, current plans call for slipping the changes in, re-testing them quickly, and making all the schedules as shown. Very heady stuff, this FCS release time.

Conclusion

The front end and back end dominated this product's schedule. The actual engineering time was only 22 weeks in a schedule that ran from June 1, 1990 through September 24, 1991. Of course, engineering proceeded in parallel early on and was running in high gear through alpha test as bugs were fixed and RFEs were granted.

With a 12 week front end and a 28 week back end, it is easy to see how product delivery time can be quite extended after a request.

Nevertheless, the deliverable is a Product: it has quality, documentation, and the force of a large company behind it. As Brooks pointed out, products are almost an order of magnitude more difficult to engineer than programs.

It is hoped that the product will provide many hours of backup pleasure to its users and generate enormous sums of money and good will for the Company.

Author Information

Jeff Polk was graduated from the University of Colorado at Colorado Springs in May, 1990. Before joining Sun Microsystems as the system administrator for the Rocky Mountain Technology Center, he had administered systems for ill-fated Prisma, Inc. in Colorado Springs and for CONVEX Computer Corporation in Richardson, Texas. His interests include automating systems administration and release control systems. Reach him via U.S. Mail at Sun Microsystems, 5465 Mark Dabling Blvd., Colorado Springs, CO 80918. Reach him via electronic mail at jeff.polk@Central.Sun.COM.

Rob Kolstad earned his Ph.D. from the University of Illinois at Urbana-Champaign in 1982. He joined start-up CONVEX Computer Corporation in Richardson, Texas as manager of the operating systems group. Five years later, he joined ill-fated startup Prisma, Inc. as manager of the operating systems group. In January, 1990, he joined Sun Microsystems as a software manager for Backup and other products. He has since joined the engineering staff and is analyzing system administration requirements. Reach him via U.S. Mail at Sun Microsystems, 5465 Mark Dabling Blvd., Colorado Springs, CO 80918. Reach him electronically at rob.kolstad@Central.Sun.COM.

References

[1] Steve Shumway, *Issues in On-line Backup*, Proceedings of the USENIX Fifth Large Installation System Administrator's Conference, San Diego, CA, pp. 81-88.

[2] Jim Engquist, *A Database for UNIX Backup*, Proceedings of the USENIX Fifth Large Installation System Administrator's Conference, San Diego, CA, pp. 89-96.

[3] Steve Shumway, *A Distributed Operator Interaction System*, Proceedings of the USENIX Fifth Large Installation System Administrator's Conference, San Diego, CA, pp. 97-104.

[4] Rob Kolstad, *A Next Step in Backup and Restore Technology*, Proceedings of the USENIX Fifth Large Installation System Administrator's Conference, San Diego, CA, pp. 73-80.

[5] Rob Kolstad & Jeff Polk, *A Faster UNIX Dump Program*, Proceedings of the USENIX Winter 1988 Conference, Dallas, TX, pp. 125-129.

Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not

Elizabeth D. Zwicky - SRI International

ABSTRACT

Many people use `tar`, `cpio`, or some variant to back up their filesystems. There are a certain number of problems with these programs documented in the manual pages, and there are others that people hear of on the street, or find out the hard way. Rumours abound as to what does and does not work, and what programs are best. I have gotten fed up, and set out to find Truth with only perl (and a number of helpers with different machines) to help me.

As everyone expects, there are many more problems than are discussed in the manual pages. The rest of the results are startling. For instance, on Suns running SunOS 4.1, the manual pages for both `tar` and `cpio` claim bugs that the programs don't actually have any more. Other "known" bugs in these programs are also mysteriously missing. On the other hand, new and exciting bugs - bugs with symptoms like confusions between file contents and their names - appear in interesting places.

The Tests

The test suite currently employs two sorts of test. First, there are static tests; files and directories with stressful names, contents, or permissions, which do not change while the program runs. Second, there are active tests; files that change while the program is running.

Static tests:

1. A file with a large hole in it.
2. A file that contains a hole and a block's worth of nulls.
3. Files with funny characters in their file names.
4. 1025 hard links to the same file.
5. 2911 hard links to different files.
6. Files with long names.
7. Symbolic links to long names.
8. Symbolic links to names with funny characters in them.
9. Unreadable and unwriteable files.
10. Unreadable and unwriteable directories with normal files in them.
11. A named pipe.
12. A device.

Active tests:

1. A file that becomes a directory.
2. A directory that becomes a file.
3. A file that is deleted.
4. A file that is created.
5. A file that shrinks.
6. Two files that grow at different rates.

Some errors occur between multiple backups or during multiple restores, and the test suite does not test for them at this point. I will mention these conditions further later.

The tests were run using a perl program which created all the static files, and then forked to modify files in one process and run the program being tested on the other. All programs were tested through a pipe, rather than having them actually create and read from tape archives. There were no compatibility tests; the archives were always being read by the program that wrote them (or, in the case of `dump`, by `restore`.) Except where specified, they were run with no options beyond those required to archive to standard output and read from standard input. A modified version of GNU diff was used to compare the original directory with the restored one.

Files With Holes In Them

Many versions of UNIX are capable of storing files with large numbers of nulls in them using what are called "holes". If a program seeks over empty portions of a file, instead of explicitly writing them, the operating system may choose to avoid actually allocating disk space for those portions. This substitution is mostly invisible to programs. Programs that read the data in the file will not be able to tell which nulls are written on the disk, and which are not. In some versions of UNIX, including POSIX-compliant versions, `stat` will return the number of blocks actually used and the size of a block; comparing this to the length of a file allows a program to determine how many holes there are, but not where they are. In other versions, not even this information is available to a program making normal use of the file system.

Programs like `dump` that read raw disk devices have no difficulty with holes, since they are reading the blocks directly. Programs like `tar` and `cpio`

that read files through the file system are almost incapable of getting holes right. There are three possible algorithms for a completely portable program to use for holes; it can ignore them, and always write nulls; it can assume that anything that could be a hole is, and never write nulls; or it can attempt to determine where the holes are by rewriting the file with explicit nulls and seeing if the file gets any longer. The last option has obvious flaws (not only is it painfully slow, but it requires a writable file and at least one block's worth of free space on the file system), and as far as I know has never been implemented[1].

Always filling in holes is the worst of the two plausible options; the most frequent fatal flaw occurs on core files in /. Core files are the most common files with holes in them, and the core files in / often have enough holes to make them the size of the virtual memory in the machine. Since / itself is often smaller than the virtual memory of the machine, a backup of / that fills in the holes may be unrestorable.

Always creating holes is somewhat better. Some programs use blocks full of nulls as a cheap way of reserving space on a disk for future use; the most common example of this these days is `mkfile` on SunOS 4.0 and above, which is used to preallocate swap files for NFS-based swapping. Replacing the nulls in a such a file with holes will have a truly unfortunate effect on performance on the machine using them for swap. It will be completely fatal if the space freed by the holes is then used by other files, since the holes cannot be filled when the machine needs the space. On the other hand, programs that do this are quite rare, and creating holes is less fatal than filling them.

Programs that are willing to sacrifice portability by requiring a `stat` that returns block size and number of blocks can always get the correct number of holes. They can't guarantee that the holes will be in the right places, and there are theoretical situations where this would be problematic - for instance, a program that intermixed seeks where it knew it would not need to put real data in, and writes of nulls to reserve space, working on a full disk, would require that the holes be where it left them[2]. I have never actually run across such a program; then again, I am not aware of any programs using this algorithm, either.

The test cases are a file which has a nearly 10 megabyte hole in it, and a file which has a block full of nulls and then a hole. The latter case tests for programs which always create holes.

[1]Barry Shein invented it solely to disprove my claim that a truly portable program working through the file system could not possibly get holes right.

[2]This example comes from Dave Curry.

Funny Characters in File Names

There are persistent rumours that some versions of `tar` are incapable of backing up files with control characters in their names. This appears to be false; on the other hand, it is quite true that newlines in file names upset `find`, which is often used in conjunction with `cpio`. It is also true that some `tar`s are incapable of writing files with funny characters in their names, even when running on operating systems that allow them.

I tested characters from octal 001 to octal 377, where possible (some operating systems do not allow characters with the high bit set). I did not test "/", because my test suite runs at too high a level to create them. "/" is completely illegal in UNIX file names, in all versions of UNIX, because it is reserved to the kernel for use as a separator. Unfortunately, it is not impossible to create files with "/" in their names. Most NFS implementations access the filesystem directly, without going through the normal kernel routines, and few of them error-check names. NFS clients running on machines like Macintoshes, where "/" is a legitimate character, may be capable of creating these files. (Sites using `aufs` to provide file service to Macintoshes are safe, since it runs as a normal UNIX process. Furthermore, on directories that it considers Macintosh-native, it performs transparent translations between "/" and ":" since ":" is a reserved separator on the Macintosh.)

By definition, no program running within the UNIX filesystem can deal with files that have "/" in their names; `dump`, like NFS, runs below the filesystem, but `restore` does not. Versions of `restore` that allow you to restore files by inode number will allow last-ditch retrieval of these files. None of the other programs tested could possibly have allowed the backup of these files, which is why I was not particularly concerned about missing that test.

Even between UNIX machines, there are file name problems. A machine running HP/UX, for instance, may be running with a 14-character file name limit. Not only will this cause problems with archives written on machines with longer limits, but it may cause problems writing archives on that machine, if it NFS mounts file systems from other machines or provides NFS service to other machines. Preferably, archive programs should avoid implementing the file system limits of the machines that they are running on while they are writing archives.

In order to achieve maximum nastiness, and also to create a very large number of files, each character appears alone as a file name, as the beginning of a file name, as the end of a file name, and as the middle of a directory name. Each directory has 10 plain files in it. For each character, the program also makes a symbolic link with the octal value as the name, and the character as the target. This test

actually turned up a nice feature in GNU tar; in verbose mode, it prints the C codes for funny characters, resulting in a much more readable listing than any other program provided.

Large Numbers of Hard Links

In the "BUGS" section of the `cpio` manual page, it states "If there are too many unique linked files, cpio runs out of memory and linking information is lost thereafter." The question is "How many is too many?" The test program creates a hard link to every file in the set with funny names (this is the primary reason for the 10 files in each oddly-named directory). It also creates 1024 extra links to a plain file, just to check for programs that have problems with large numbers of links to the same file.

Long File Names and Symbolic Link Targets

The `tar` manual page documents a 100-character path length limit; the `cpio` manual page under SunOS 4.1 cpio documents a 128-character path length limit. The `pax` manual page more vaguely admits "there are restrictions on the length of pathnames stored in the archive" because of restrictions on the formats it uses. In `tar`'s case the 100-character restriction is indeed a firm upper limit because of the format; in `cpio`'s it is not, and varies by implementation. (Most versions of `tar` do not make it all the way to the 100-character limit, so there is minor room for variation by implementation there, too - there are also people intentionally running mutant `tar`-like programs that produce incompatible tapes with a longer limit.)

100 characters may be a reasonable limit on systems with a 14-character file name limit, where you have to type all the characters in every file name yourself. On systems where file names may go up to 255 characters, and filename-completing or icon-based shells are available, users may overrun 100 character limits quite frequently.

There are two constraints on name length; a limit on how long an individual component of the path may be, and a limit on the total length of the path. On BSD-based UNIX implementations, these are usually 255 and 1023 characters, respectively. What many people fail to take into account is that these are both per-filesystem limits; your 1023 character pathlength limit starts counting at the beginning of the name of the mount point, not at /. Simply allocating a 1024 character buffer - or even finding the maximum path length and allocating a buffer that size - does not guarantee you will actually be able to fit any file name, starting from /. Nothing I've found, including `dump` and `restore`, actually manages to archive a file with a name that is the maximum allowable. To be fair, under SunOS 4.1, `fsck` considers a file with a name over 1021 characters to be an error, also.

Many archive formats silently implement a limit on the length of the target of a symbolic link, either by failing to write the link at all, or by truncating it. Some will notify you that they were unable to archive the link. So far, I haven't found a program with a limit that also correctly documents what the limit is.

Unreadable and Unwriteable Files and Directories

An archive program that is being run by root on a local disk should have no problems with file permissions. Users running their own archives, or root running over NFS, may run into permission problems. Some permission difficulties are well-known - for instance, the `cpio` manual page recommends running `find` depth-first so that files will appear on the tape before their containing directories, in order to avoid difficulties with directory permissions.

These tests are run as the user who owns the files; theoretically, the archive program could have forcibly read the unreadable files by changing the permissions. Whether this would be a good thing or not is an open question.

Named Pipes and Devices

Some archive programs, like `tar`, archive only "normal" files. Others will pick up special files as well. The most spectacular failures here, under Encore Mach, are not actually the fault of the archiver; that version of the operating system turns out to crash whenever you do almost anything to a named pipe, including remove it.

A program that does not archive special files is going to be a severe annoyance if you need to restore an entire system, since you will have to build at least /dev by hand.

Hard Links to Directories

I ran some tests with hard links between directories in place. It is not a standard part of the test suite, because it wreaks even worse havoc than the normal tests; it becomes very hard to determine exactly what is causing things to fail. Furthermore, the number of non-archive programs that fail becomes a real trial. All the archive programs fail, one way or another, including `dump`. Hard linked directories are an error, and will be caught and removed by `fsck`; on the other hand, very few people run `fsck` before every backup. In most cases, the hardlinked directories are skipped. Systems using `find` ended up skipping the hardlinked directories, but achieved this effect by being massively confused, providing dozens of confusing and irrelevant error messages (mostly complaining that the files in the hardlinked directories didn't exist). `tar` actually succeeds in backing them up - the hardlinks disappear in the restored version.

Active Tests

The active tests are not as complete as the passive ones, for several reasons. First, I wasn't able to run the active tests on all of the machines that the passive tests were run on; the passive tests could be run on any machine that could NFS mount a directory, but the active test required a functioning perl. Perl is probably capable of running on all the machines, but I wasn't in a position to install it on all of them in time to run the tests. Second, many of the active tests only produce errors if the timing of the events happens to be just right. Third, there are cases in which programs provably get active tests wrong, but silently compensate for the error in most cases. I intend to work up a more effective test suite (and to install perl on the machines that currently don't have it). Meanwhile, I will sketch the problems tested for.

Programs that do not go through the file system, like `dump`, write out the directory structure of a file system and the contents of files separately. A file that becomes a directory or a directory that becomes a file will create nasty problems, since the content of the inode is not what it is supposed to be. Restoring the backup will create a file with the original type and the new contents.

Similarly, if the directory information is written out and then the contents of the files, a file that is deleted during the run will still appear on the tape, with indeterminate contents, depending on whether or not the blocks were also re-used during the run

All of the above cases are particular problems for `dump` and its relatives; programs that go through the file system are less sensitive to them. On the other hand, files that shrink or grow while a backup is running are more severe problems for `tar`, and other file system based programs. `dump` will write the blocks it intends to, regardless of what has happened to the file; if the file has been shortened by a block or more, this will add garbage to the end of it, and if it has lengthened, it will truncate it. These are annoying but non-fatal occurences. Programs that go through the file system, on the other hand, write a file header, which includes the length, and then the data. Unless the programmer has thought to compare the original length with the amount of data written, these may disagree. Reading the resulting archive - particularly attempting to read individual files - may have unfortunate results[3].

Theoretically, programs in this situation will either truncate or pad the data to the correct length. Many of them will notify you that the length has changed, as well. Unfortunately, many programs do not actually do truncation or padding; some programs even provide the notification anyway. In many cases, the side reading the archive will compensate, making this hard to catch. SunOS 4.1 `tar`, for instance, will warn you that a file has changed size, and will read an archive with a changed size in it without complaints. Only the fact that the test program, which runs until the archiver exits, got ahead of `tar`, which was reading until the file ended, demonstrated the problem. (Eventually the disk filled up, breaking the deadlock.)

Untested Problems

The test suite in its current states looks only at problems that arise within a single archive. There are other problems that arise when you use programs designed for single archives to do multi-level dumps, attempting to pick up files that have changed since the last pass.

Programs that work through the filesystem (including `tar`, `pax`, and `cpio`) modify the inode access time. This is also the only time that is changed when changes are made to the permissions on files, so systems using these programs cannot also pick up changes to permissions. (This is the same problem that `rdist` exhibits.)

Programs that are not designed for multiple-level backups (again, including `tar`, `pax`, and `cpio`) mark only existing files on their archives. Since they don't know about past state, they cannot mark files that have been deleted or renamed since previous runs. This is not a problem as long as you are making backups, but if you need to restore them, the result is likely to be more files than space; every file that ever got onto a backup will be restored, and every renamed file or directory will be present twice, once under each name.

Other Warnings

Most of the things that people told me were problems with specific programs weren't; on the other hand, several people (including me) confidently predicted correct behavior in cases where it didn't happen. Most of this was due to people assuming that all versions of a program were identical, but the name of a program isn't a very good predictor of its behaviour. Beware of statements about what "tar" does, since most of them are either statements about what it ought to do, or what some particular version of it once did. Also watch out for people who state that they've never had a problem backing up files of some type; what you care about is whether you can back them up and then restore them. You can back up files with "/" in them, for instance, until the cows come home, without problems. You just can't restore them.

Don't trust programs to tell you when they get things wrong either. Many of the cases in which things disappeared, got renamed, or ended up linked to fascinating places involved no error messages at

[3] "cpio out of phase: get help!" springs to mind

all.

Some programs had peculiarities which were not consistent, or weren't relevant to the tests. `pax`, for instance, worked in some situations only if run in verbose mode. Under Encore Mach, `tar` core dumped while running on the funny names on one run (complaining multiple times of an unknown write error 70, and finally saying "HELP" and dying). This behaviour did not reappear on later runs. It also tended to produce error messages in blocks, with first the content part of 20-40 messages, and then the "tar:" from the beginning of all of them. Not all programs got multiple test runs, so in some cases I may have hit or missed intermittent problems.

Although the test suite runs lots of tests, it is by no means exhaustive. For instance, since I set the suite up I have heard it suggested that some versions of `cpio` convert symbolic links to hard links where possible. The test suite does not create any valid symbolic links, so this problem would not show up. There are also problems that are extremely rare – for instance, when a `dump` tape begins with the continuation of an inode which is another `dump`.

The testing directories tend to turn up bugs in unexpected places; anything that tries to walk directory trees is probably history. That includes not only `find`, but also `du`, `rm -r`, `diff`, and even `gnudiff`. Painful experience taught me that while I was working in the testing directories, I needed to turn off the `cd` alias that showed the current directory in the prompt. Not only do prompts longer than a few hundred characters cause the version of `tcsh` I run to core dump, but the alias I use won't let me out of a directory with a space in its name. I also learned to delete the test directories promptly, to avoid the complaints of my co-administrators about the mail generated by dying `cron` jobs, and the censure I got for having a directory that `du` found an extra 300 megabytes in.

Conclusions

These results are in most cases stunningly appalling. `dump` comes out ahead, which is no great surprise. The fact that it fails the name length tests is a nasty surprise, since theoretically it doesn't care what the full name of a file is; on the other hand, it fails late enough that it does not seem to be an immediate problem. Everything else fails in some crucial area. For copying portions of file systems, `afio` appears to be about as good as it gets, if you have long file names. If you know that all of the files will fit within the path limitations, GNU `tar` is probably better, since it handles large numbers of links and permission problems better.

Looking at tables in Appendix A, it's easy to fall into a deep depression (after the initial incredulity wears off). It's worth remembering that most people who use these programs don't encounter these problems.

My testing programs are available for anonymous ftp from ftp.erg.sri.com, with some warnings. Chief among them is the warning not to run them on a machine that's critical; even if the tests don't crash it, and they probably won't, you'll immediately be blamed for anything that goes wrong.

Author Information

Elizabeth Zwicky is a system administrator for the Information, Telecommunications, and Automation Division at SRI International, where she relieves tension by torturing innocent file systems. Reach her via U.S. Mail at SRI International; 333 Ravenswood Avenue; Menlo Park, CA 94025. Reach her electronically at zwicky@erg.sri.com.

Appendix A: Tables of Evaluations

	Large Hole	Deceptive Hole
Tar (all known versions except Gnu)	Filled in	Filled in
Gnutar 1.10 with -S	Correct	New hole created
Cpio (all known versions)	Filled in	Filled in
Pax (SunOS 4.1)	Correct	New hole created
Afio (SunOS 4.1)	1 block of nulls filled in	Correct
Dump (all known versions)	Correct	Correct

Table 1: Holes in Files

	File Names	Symbolic Link Targets
tar (SunOS 4.1, HP-UX 7.0, Irix 3.3.2)	Correct	Correct
tar (Encore Mach 1.0, Mt Xinu Mach)	Files with 8th bit set missing	Correct
pax -x ustar (SunOS 4.1)	Correct	Correct
Gnutar 1.10	Correct	Correct
find | cpio (SunOS 4.1, HP-UX 7.0)	Files with newline missing (problem is in find)	Correct
find | cpio (Encore Mach 1.0)	Files with newline or 8th bit set missing	8th bit stripped
find | cpio (Mt Xinu Mach)	Files with newline or 8th bit set missing	Correct
pax -x paxcpio (SunOS 4.1)	Correct	Correct
find | afio (SunOS 4.1)	Files with newline missing (problem is in find). Some files became executable; some directories became normal files. The problem does not appear to be funny characters, since the affected directories included "151(i)dir" and others that had only normal characters in the name.	7 symbolic links converted to regular files. Their original targets were "!", "#", "K", "L", "N", "O", and "T".[4]
find2perl -cpio[5]	Correct	Correct

Table 2: Funny Characters in Names

[4]This doesn't seem to be a funny character problem, but I have no idea what it is.
[5]find2perl: 2-line modification to quote file names.

	Unique links	Multiple links
tar (SunOS 4.1, Encore Mach 1.0, HP-UX 7.0, Irix 3.3.2)	Correct	Correct
Pax -x ustar (SunOS 4.1)	256	Correct
Gnutar 1.10	Correct	Correct
find | cpio (SunOS 4.1)	All linked files are still linked, but only 2268 of them correctly - the remainder are crosslinked among each other, with link counts up to 8	Gained 2 extras out of the unique links
find | cpio (Encore Mach 1.0, Mt. Xinu Mach, HP-UX 7.0)	Correct	Correct
pax -x paxcpio (SunOS 4.1)	Two showed up linked to the multiples; rest were not linked at all	The right number, but 2 of the wrong files, 2 of the originals omitted
find | afio	One linked to multiples; some not linked at all; some of the linked ones apparently linked to the wrong places; some of the linked ones added execute permission.	Right number of links, but one of them to the wrong place, and the last one of the originals omitted
dump (SunOS 4.1)	Correct	Correct

Table 3: Large Numbers of Hard Links

	File becomes directory	Directory becomes file	File is created
Tar (SunOS 4.1)	Directory	File	File exists
Gnu tar 1.10	Directory	File	File doesn't exist
afio	Directory	File, but with 2 links	File exists
find2perl -cpio	Directory	File	File exists

	File is deleted	File shrinks	File grows
Tar (SunOS 4.1)	File doesn't exist	Apparently OK	Writes until end-of-file
Gnu tar 1.10	File doesn't exist	Notes that file has shrunk, but claims it has shrunk by 0 bytes	OK
afio	File doesn't exist	OK	OK
find2perl -cpio	File doesn't exist	cpio became out of sync and exited	Untested

Table 4: Active Tests

	Pathname length	Symbolic link length
tar (SunOS 4.1, Encore Mach 1.0, Mt. Xinu Mach, Irix 3.3.2)	99 characters (100 claimed in man page)	98 characters
tar O (HP-UX 7.0)	99 characters	98 characters (100 claimed in man page)
tar N (HP-UX 7.0)	125 character pathname (100 character filename) (256 claimed in man page)	98 characters (100 claimed in man page)
Gnutar 1.10	86 characters; overlength files are put in ./MaNgLeD<filenumber>, with warning	99 characters; overlength silently truncated
pax -x ustar (SunOS 4.1)	99 characters; at 100 writes the file but appends ''000644'' to the end of the name. Does not produce an error message until much later. In directories above the pathname limit, puts files on the tape as ./filename, truncating the filename at 100 characters and apppending 000644	99 characters; above 100 characters appends ''ustar'' to the end of the link but writes it anyway without comment.
find | cpio (SunOS 4.1)	Writes all the files, but gets confused at 258 characters and complains that it cannot change mode on a file that has the first 258 characters of the path name, plus the contents of the file, as its name (128 claimed in man page)	Correct (tested to 256)
find | cpio (Encore Mach 1.0, Mt Xinu Mach)	Writes all the files, but gets confused at 256 characters and complains that it cannot change mode on a file that has the first 256 characters of the path name, plus the contents of the file, as its name (128 claimed in man page)	Correct
pax -x paxcpio	253 characters, complains about a damaged archive	255 characters
find | afio (SunOS 4.1)	Find complained about pathname too long on the long directory; using find2perl, afio succeeded	Correct
dump (SunOS 4.1)	1021 characters (OS limit is 1023; the difference appears to be dump appending ''./'' to the pathname); restore gives up even earlier, at 1015 characters	Correct

Table 5: Long File Names and Long Link Names

	Unreadable file	Unreadable directory	Unwriteable file	Unwriteable directory
tar (SunOS 4.1)	Missing	Exists, with contents, is now drwxr-xr-x	Correct	Exists, with contents, is now u+w
tar (Encore Mach 1.0)	Missing	Exists, new permissions, no contents	Missing	Exists, with contents, is now u+w
tar (Mt Xinu Mach)	Missing	Exists, is now drwxr-xr-x	Correct	Exists, with contents, is now u+w
tar (HP-UX 7.0)	Missing	Exists, new permissions, no contents	Correct	Exists, with contents, is now u+w
tar (Irix 3.3.2)	Missing	Exists, no contents	Correct	Exists, no contents
pax -x ustar (SunOS 4.1)	Missing	Causes the reading pax to core dump	Correct	No contents
Gnutar 1.10	Missing	Exists, wrong permissions[6], no contents	Exists, wrong permissions[6]	Exists, with contents, wrong permissions[6]
cpio (SunOS 4.1, Mt Xinu Mach)	Missing	Exists, correct permissions, no contents	Correct	Correct
pax -x paxcpio (SunOS 4.1)	Missing	Causes the reading pax to core dump	Correct	No contents
find | afio (SunOS 4.1)	Missing	Exists, correct permissions, no contents	Correct	No contents
dump (SunOS 4.1)	Correct	Correct	Correct	Correct

Table 6: Difficult Permissions on Files and Directories

	Named Pipes	Devices
Tar (SunOS 4.1)	Missing	Missing
Tar (Encore Mach 1.0)	Machine crashes	Missing
Tar (Mt. Xinu Mach)	Correct[7]	Correct
Tar O (HP-UX 7.0)	Missing	Missing
Tar N (HP-UX 7.0)	Correct	Correct
Pax -x ustar (SunOS 4.1)	Correct	Correct
Gnutar 1.10	Correct	Correct
Cpio (SunOS 4.1, HP-UX 7.0)	Correct	Correct
Cpio (Encore Mach 1.0)	Machine crashes	Correct
pax -x paxcpio (SunOS 4.1)	Correct	Correct
Afio	Became regular file	

Table 7: Special Files

[6]Gnutar warned that it was changing the permissions; the only change was the application of the current umask.

[7]The OS does not support named pipes, so creation fails, but the correct data is present in the archive and tar attempts the creations

Backups Without Tapes

Liza Y. Weissler - The RAND Corporation

ABSTRACT

This paper describes an optical backup system in use at the RAND Corporation over the past 18 months. Incremental backups of Sun fileservers and diskful clients are written to optical disc on Epoch InfiniteStorage fileservers. Basic design, advantages and disadvantages of the system are described, as well as some of our experiences with the Epochs in general.

Introduction

This paper describes an optical incremental backup system in use on the RAND Computer Information Systems (CIS) networks of UNIX systems over the past 18 months. The RAND CIS networks include 21 Sun-3 and Sun-4 NFS servers and 37 diskful clients requiring backups, with a total capacity of about 90 gigabytes. Most of the user filesystems are fairly volatile, with incremental backups averaging 10% of a given system's capacity nightly.

The old backup system in use at RAND was "invented" over ten years ago by an unnamed system programmer. The original design was intended to backup one, perhaps two systems with limited disk space to local 800 bpi 9-track tape. The system of scripts grew by accretion, as a number of system administrators patched the scripts to support use of 1600/6250 bpi tapes, dumping of Sun ND client partitions, dumping to remote tape drives, use of 8mm exabyte tapes, differences between VAXen and Suns, and so forth. The result was a hodgepodge of scripts that was not aesthetically pleasing, nor particularly efficient; but while the capacity of RAND systems remained relatively small, the scripts worked to everyone's satisfaction, and there was little incentive to rework them.

In 1989, however, the RAND networks began to grow steadily - in that one year, our network capacity nearly doubled - and the shortcomings of the old dump scripts became increasingly apparent. There were two distinct sets of backup scripts, one to handle backups to local tape, the other to remote tape. The scripts required a separate exabyte tape, per system, per day (6 incremental, 1 full per week), and were rather graceless upon reaching the end of an exabyte tape. As a result, we wasted a great deal of tape on our incrementals, and overran the tapes when dumping our larger (i.e., over 2 gig of storage) fileservers. The volume of tapes to catalog/store/cycle-to-offsite storage ceased to be manageable by our tape librarian.

Worst of all, the backups and restores were very slow. Dumping 50+ systems to four, perhaps five exabyte tape drives required a full shift of operator time each day. Restoring files was a frustrating process, as forward-spacing on the tapes was slow, mistakes were time-consuming, and the users' general tendency toward sloppy reporting of when files were created/lost resulted in the operations staff spending more time on restores than we could afford. Clearly our backup procedures needed retooling.

Enter the Epoch

In late 1989, RAND acquired its first of two Epoch InfiniteStorage servers. Epoch servers support the InfiniteStorage Architecture (ISA), a hierarchical storage architecture that more-or-less transparently integrates removable optical discs, both erasable (EO) and WORM, as back-end storage for magnetic disks.

As an Epoch filesystem on magnetic disk reaches a *high water mark*, (usually 95% of capacity, although this is tunable), files written to that filesystem are *staged*, or transferred, to an appropriate optical disc (the *current staging volume*) according to a staging algorithm based on each file's size and age. Such staging will transfer files sufficient to push the filesystem's usage down to a *low water mark* (usually 88% of capacity). Subsequent accesses of staged-out files result in their being written back to magnetic disk. From an end-user's point of view, the filesystem is seemingly of "infinite"[1] size, and the stage-in/stage-out process is usually transparent[2].

Our original intent in purchasing the Epoch servers was to provide our users with more storage/archival space. We soon began to look for other ways to exploit the capabilities of the Epoch systems in our environment; incremental backups seemed to be a good application. We decided to have our systems write compressed dump images to an Epoch filesystem, and stage the compressed dump images from the Epoch filesystem to optical disc. Our unwieldy collection of backup scripts were thus

[1] "Infinite", alas, is still a relative term, as the size of a given file is still limited by the size of the optical media.

[2] If the file one wishes to access is on an optical disc that is "out of the jukebox", i.e., not in the Epoch's Optical Library Unit (OLU), then the stage-in process is decidedly *not* transparent, as the user's process blocks until someone notices that the Epoch requires an optical mount.

replaced by the following files:

daily.backup – Simple *csh*(1) script that runs on each fileserver and diskful client. *daily.backup* creates an appropriate directory on the Epoch; determines which filesystems, if any, are to be backed up; determines the dump level; runs *dump*(8) on each filesystem; runs *compress*(1) and *epfilesplit*(1) on each resulting dump file[3]; and finally writes the results to the Epoch. After each system's dumps are complete, the compressed dump images on the Epoch are forced out to optical disc (thus freeing more magnetic space for subsequent dumps) with Epoch's *epstage*(1) command.

daily.backup.fe – A *csh*(1) front-end to *daily.backup*. This script is run once nightly via *cron*(8) from the "oper" user's crontab; it runs *daily.backup*, captures the output in a dumplog on each system, copies the output to a central Operations directory, and sends notifies interested persons via email when dumps are complete.

backup-config – File used by *daily.backup.fe* to control exactly when *daily.backup* will begin. *daily.backup.fe*, as noted above, is run from oper's crontab, and our oper crontabs are maintained via *rdist*(8.) The *backup.config* file allows us to delay the start time of *daily.backup* on each system (thus we avoid having all of our systems attempting to write to the Epoch simultneously) while maintaining identical crontabs everywhere. Entries in this file are in the format "*hostname n*", where *n* is the number of seconds to sleep before running *daily.backup*. This file is generated periodically based on information in "site" databases maintained by our system administration group.

orestore – An operator front-end to *restore*(8) allowing operations to run interactive restores from the compressed dump files. This saves our Operations staff from (a) needing to know exactly where the compressed dump images live, and (b) having to type in the rather gory command line in order to run a restore.

daily.backup, as noted, *epfilesplit*'s the compressed dump files; the maximum file size we allow is 25 meg. There are two reasons for this limit:

(1) As noted above, an Epoch may be called an "InfiniteStorage" system, but a single file is still limited to what will fit on one side of an optical disc[4]. The two models of Epochs currently in use at RAND both use 5-1/4" erasable opticals with an upper limit of about 300 meg.

(2) One is further constrained by the size of the Epoch filesystem to which the dump images are being written before they stage to optical disc. The *epfilesplit* limit must be low enough to allow eight to ten systems to simultaneously write chunks of compressed dumps to the filesystem without overrunning the available space.[5] Our Epoch backup filesystem is currently 500 meg; we've set our *epfilesplit* limit rather conservatively.

Dump images for each system are maintained on the Epoch for two weeks. Old dump images are removed from the filesystem via a nightly *find*(1) process, and the resulting "stale" space on the optical discs is reclaimed during the Epoch's nightly compaction procedure.

Experiences

As one might suspect, using optical systems for backups has its advantages and disadvantages.

The primary advantage is that our incremental backups now run completely unattended, provided that we keep several "available"[6] opticals in the Optical Library Unit to be used should the current staging optical fill to capacity. Unattended backups have saved our operations staff hours of labor, allowing our computing center to run with fewer personnel, and to have those personnel concentrate on other functions. File restorations (via the *orestore* script) are speedier, and the number of tapes to catalog/store/recycle has dropped off dramatically[7].

From an administrative viewpoint, however, there are a number of concerns. First, we're relying on optical media to be minimally as "safe" as exabyte tape, if not better. We continue to experience occasional problems with our optical media, such that the discs must be "dry-cleaned" (and failing that, "wet-cleaned") before data can be retrieved in its entirety; to date, though, none of our opticals (out of a pool of about 150) have been irretrievably damaged.

Second, our use of the Epochs gives us a single point of failure for both backups *and* restores on each of our networks. The optical discs are (currently) not readable on any other system; in the

[3] *epfilesplit* is an Epoch facility for splitting non-ascii files.

[4] Future releases of Epoch OS will support *composite optical volumes*, where a single file can span multiple opticals.

[5] Multiple processes overrunning one's magnetic storage on an Epoch system can cause major headaches, especially if the Epoch thinks there are no viable "candidate" files for staging on the filesystem.

[6] Formatted, but not allocated.

[7] Which is not to say, though, that we've saved any money on media. Under our old backup scheme, we typically used about 1000 exabyte tapes per year, at a unit cost of $6. Running backups to the Epochs, we've reached a "steady state" of using 45 or so optical discs, at a unit cost of about $300.

case of a disaster our ability to completely restore one or more filesystems beyond the last full tape backup could be seriously delayed. Temporary loss of the Epoch is less crucial for the backups themselves; should the Epoch be down at the *cron*-appointed backup time, the Operations staff can easily restart the backup scripts on all of the systems once the Epoch is back up. Major downtimes require that the Operations staff regroup and run backups of all systems to tape - an inconvenience, but not a serious problem.

Last, since the backups are unattended, there could be a tendency toward considering the backups "out of sight, out of mind". We're relying on our operations staff (usually the midnight-08:00 shift) to check the dumplogs for errors and take appropriate action.

Future Plans

Although the backup-to-optical scheme in use at RAND works, and works reasonably well, it will probably remain in production for at most one more year. We're actively investigating other backup strategies and commercial backup products that are more robust and provide more features, including Epoch's NFS Backup and Renaissance Backup products.

Author Information

Liza Y. Weissler is the Project Leader for UNIX System Administration in the RAND Corporation's Computer Information Systems department, where she manages a group that administers over 200 Sun workstations. She received an MSLS from the University of Southern California in 1984, and worked at System Development Corporation as a system administrator and technical writer before joining RAND in 1986. Reach her via U.S. Mail at The RAND Corporation; 1700 Main Street; Santa Monica, CA 90407-2138. Reach her electronically at liza@rand.org.

Configuration Control and Management

Ed Arnold & Craig Ruff - National Center for Atmospheric Research

ABSTRACT

It has become apparent that a Configuration Control System (CCS) has become necessary to support NCAR's computing environment. With the presence of complex networked systems such as two Cray Y-MPs running UNICOS, minimization of downtime and speedier disaster recovery have become more important.

This paper describes both planned work-in-progress, and an implementation of automatic filesystem auditing. The impetus behind work-in-progress is to be able to install, de-install, and verify major parts of a system (binary software packages), with a high degree of confidence that everything needed by the package is in-place and has not been corrupted. The concept of a "release or package description" has already been applied to systems as a whole; this provides continuous monitoring of system content to improve system integrity.

Introduction

There is at the current time, no widely-available public-domain software for controlling, managing, and auditing the configuration of a UNIX system in a binary sense. This paper describes both work-in-progress (plans we have made for work in this area), and work already done which informs us of changes in filesystem configuration.

The Problem

At NCAR, the direct staff in various operating divisions utilize dozens of Unix machines as part of the array of tools they need to support a staff of scientists, and external users, in carrying out the basic mission of research in the atmospheric sciences. The heart of NCAR's computing complex now consists of two models of Cray Y-MP machines running UNICOS, which demand better management techniques than have been used in the past.

Good management of these resources requires that we be able to accomplish a number of goals:

- Of a number of software packages which either come to us from outside (e.g. X11) or are internally-written (e.g. NCAR Graphics), we should be able to determine which are present on a system, and whether they are in working order (components not missing or corrupted).
- We should be able to determine which version of a package is installed, without having to contact the author or primary maintainer. Some packages may not contain an SCCS/RCS version string in the binaries.
- We should be able to install or re-install a package on a system without having to give away the root password to developers or other parties.
- We would like to keep master copies of packages on a central repository. In our case, that is the NCAR Mass-Storage System (MSS), a three-tiered repository (disk, automatic cartridge library, and off-line cartridges) which already holds regular backups from many machines in our complex.
- We need to be able to create an audit trail of what changes have been made to the most critical systems.
- Treating each whole system as a "package", we would like to be able to automatically audit changes in the content of the package on a periodic basis to determine if hardware failure, staff error/non-communication, or break-in has compromised the package as a whole.
- Of course, we must be able to accomplish these goals on a variety of Unix equipment, including Cray, DEC, HP, IBM, and Sun, so portability must be taken into account. At our site, perl has been found to be useful in this regard.

The Solution

The solution we are pursuing is a Configuration Control System. Most of the system discussed here is still in planning stages. However, we have for some time been able to audit filesystem content to track changes.

Package Attributes

Specification of a software package to the CCS would involve at least the following data objects in the repository sub-trees used to describe packages.

PkgName would describe the overall name of a package of program(s) and data/configuration file(s) required by those programs.

PkgRev would define the revision number of the package itself.

ProcArch would define a family of processor architectures on which this package would run.

Os would define a an OS on which this software should run (UNICOS, SunOS, etc.).

OsRev would define OS revision levels on which this package would run.

Hostname would necessarily be included if the package included hardware or other elements unique to a single machine.

A tree structure is not the best way to organize these attributes, however, the above top-to-bottom ordering seems reasonable in view of our previous experiences at storing OS software on our MSS, which have too frequently included not enough info about processor architecture. In practice we would probably collapse PkgName/PkgRev and OS/OsRev into a directory level together.

Package Files

At this time, it appears that three data files would be necessary to implement all the functions required by the CCS. These are a **PkgAr** file (tar or cpio) that contains the actual software itself, a **PkgDesc** which would contain a description sufficient to verify the software without having to extract it from its archive file, and a **PkgLog** that contains a record of the functions carried out against a given package.

Functions

Given that an administrator, package maintainer, or end-user had supplied the package attributes mentioned above, the CCS must be able to perform the following functions to meet our requirements:

Verify would verify the configuration of the specified package against the PkgDesc file. Usually this would be controlled by the **stringency** of parameters in the description file, which might not be the same for every package, or every file of every package.

Install would install a package on a system, perhaps combined with a **Verify** afterwards.

Remove would remove a package from a system, directed by the information contained in the PkgDesc file. The CCS would of course verify that the installed software matched the PkgDesc file before proceeding.

Display All would, given only the name of a package, display all the versions of a package, processors for which is is available, etc. from the central repository.

Display would display the version of the package that is currently installed on the current machine. Since specification of the package name alone may not be sufficient to identify file(s) and thus RCS/SCCS version headers in the package, this essentially requires that we be able to fetch a PkgDesc file from a repository, create that same description for files in that package, and determine if there is a match.

Require is essentially the same as a **Display** which fails. Users sometimes have requirements in their jobs to use only a specified version of a specified package, and this will allow them to terminate their job early if the specified package is not present for any reason.

Build would, given the necessary package attributes, create the package data files and place them in the repository for later use by an **Install.**

ViewLog would fetch and display the log of functions that have been carried out on a given package.

Vet would look at the package at a binary level to warn of possible security problems, e.g. install of .rhosts files, install of setuid/setgid binaries, etc.

Other functions could be constructed from the basic ones defined above.

Automatic Filesystem Auditing

The bulk of what is discussed in this paper, i.e. management of software packages from a central repository, on multiple machine architectures, operating system types, etc., represents work-in-progress. However, we have been doing work in the area of **configuration description/automatic filesystem auditing** for a while. By representing the system portions of a machine's filesystems in a PkgDesc file, we build an extra level of confidence that hardware problems, break-ins, staff errors, and staff non-communication, may be detected before they have a chance to negatively affect machine operation.

The auditing operation is complementary to our automatic backup system, as well as other auditing operations such as the COPS security system. There are two functions, done by perl scripts. A **Create** builds a PkgDesc file in the backup logging area local to each machine, each of which is arranged on date. A **Diff** applies a Unix "diff" against PkgDesc for the past two dates, or specified dates, and mails this to an administrator if there is any output.

The low-level building of the PkgDesc is done by a C program called "rls." This program bypasses the obvious shortcomings of "ls", most notably, output that is not real regular, inability to accept a file list on an input pipe, and lack of any measure of file content (checksum or CRC). Its output is a compact ASCII description that is suitable for processing by awk or perl. It contains one feature essential to a task of this type: control over the **stringency** of parameters generated, both globally and per wildcard filename. For instance, checksumming of files like client swap areas is a waste of processor

resources; and checksumming of files like (constantly-changing) log files is unnecessary, since we are interested mainly in their protection and ownership. By changing the stringency requirements on given files or sets of files, we have reasonable assurance that reported changes will be real changes we are interested in, not false hits.

The idea of a compact configuration description has also been found useful for saving the vendor-supplied state of a system. This can later reduce the amount of manual work that needs to be done to identify local changes in a system prior to system upgrade.

Conclusions

Based on experience at NCAR with trying to keep multiple software packages in operation on proliferating machines, and the obvious benefits of both automatic backup (*Automatic Unix Backup in a Mass-Storage Environment*, Usenix Winter Proceedings 1988) and automatic filesystem auditing, the next logical step for our organization is to build some type of Configuration Control System. Without it, we don't believe we can adequately protect NCAR's computing environment, nor insure its consistency.

Author Information

Ed Arnold cut his teeth on Unix at StorageTek, after several years in the telephony business. At Storage, he developed production equipment and diagnostic disk drivers. Since coming to NCAR in 1983, he has been doing system administration and programming, and living the movie *Gaby* after hours. Reach him via U.S. Mail at NCAR, POB 3000, Boulder CO 80307-3000. Reach him electronically at era@ncar.ucar.edu.

Craig Ruff is Group Head of the System Administration and TAGS (Text and Graphics Server) Group within NCAR's Scientific Computing Division. After graduating from college in 1984, he has done a variety of Unix administration, network, system and application programming for several companies. Feeling the need to escape from the morass of U.S. Government computer-related program development, he happily surfaced at NCAR in 1987. When not chained to a workstation, he likes to listen to music, garden and hike in the mountains with his dog. Reach him via U.S. Mail at NCAR, POB 3000, Boulder CO 80307-3000. Reach him electronically at cruff@ncar.ucar.edu.

hobgoblin: A File and Directory Auditor

Kenneth Rich & Scott Leadley - University of Rochester

Consistency is the hobgoblin of small minds, small statesmen ... Ralph Waldo Emerson

ABSTRACT

hobgoblin is a language and an interpreter. The language describes properties of a set of hierarchically organized files. The interpreter checks the description for conformity between the described and actual file properties. The description constitutes a model for this set of files. Consistency checking verifies that the real state of these files corresponds to the model, flagging any exceptions. **hobgoblin** can verify conformity of system files on a large number of systems to a uniform model. Relying on this verification, system managers can deal with a small number of conceptual models of systems, instead of a large number of unique systems. Also, checking for conformity to an appropriate model can enhance system reliability and security by detecting incorrect access permissions or non-conforming program and configuration files.

Motivation

A fair amount of the work involved in managing a system is concerned with managing the files in a system. Doing this effectively for a large number of systems has to involve dealing with abstract models instead of myriads of low level details unique to each system. The system manager must therefore have confidence that an abstract model accurately reflects and perhaps even influences the system he or she is working on. Unfortunately, there are no tools available that specifically verify the conformity of a system to such a model. Nor do we have any models to check against.

Dealing with an abstract model sounds high-minded and noble, but how does it relate to the everyday practice of system management?

Systems are self documenting. On most systems we assume system files have the properties that they ought to have. The only definition or model external to the system is usually in the system manager's head. Given the large numbers of files, the necessary differences in file attributes from system to system, and the possible negative consequences of experimentation, it is not surprising that many system managers adopt an, "if it ain't broke don't fix it" attitude. This leads to sometimes wild inconsistencies between systems that should operate identically, system guano (unnecessary files and changes to files left over from previous system management work), ad hoc enforcement of proper access permissions for directories and configuration files, and uncertain detection of system security violations such as unauthorized additions and modifications to system utilities.

In addition, software distribution tracking is usually either not done, done informally, or done by examining the audit records or configuration files of a software distribution system. If there is any doubt that any portion of a software package is incorrect, the best solution that can usually be achieved is to reinstall the software package or compare a fixed set of attributes (e.g. modification date and size) to the attributes of a known good copy of the package.

We have a small staff and hand out root user privileges to trusted co-administrators with specific areas of responsibility. This trust has sometimes been abused. Rather than become control freaks, thereby increasing both our work load and turn-around time for filling user requests, we would like to trust but verify.

Prior Art

There are a few tools that address some aspects of this problem.

- **vcheck** is a utility distributed with UniSoft system source code. It can describe, check, and correct the mode, user and group of a file. It is, however, not available separately.
- **rdist** certainly deserves mention, but it is limited to verifying modification date, size and optionally byte-for-byte equivalence. It does not verify the access permissions, owner or group of a file on a constant basis. It requires a full and correct copy of the files being checked, and has no tolerance for necessary local differences.
- **find** is another well-known utility for filesystem checking. It has a hard time checking specific characteristics of specific files. Its nice selection of operators inspired the creation of much of **hobgoblin**'s collection of internal checkers.
- **COPS**, the Computer Oracle and Password System by Dan Farmer at Purdue, is another

tool that we use that overlaps **hobgoblin** territory. We think **hobgoblin** might keep better track of the detail in filesystems and is faster and more configurable than COPS.

- Sun's **auditd** might be of use in monitoring systems for some installations. **hobgoblin** only monitors this activity after the fact through any residue that the activity leaves in the filesystem. However, because of disk space required by auditing and programmer time constraints, we have elected to not investigate the audit daemon any further until audit management tools appear and disk gets a lot cheaper and more abundant. We have found that system usage accounting [2] can fill some of the critical functions of the audit daemon with less system and disk overhead.

Keeping It Simple

Hobgoblin does not directly make a system more useful to the end user. Besides, it adds Yet Another Language to the Unix tower of Babel. We want a language that is as simple, understandable, and easy to use as possible. Some characteristics of such a language are: a regular syntax, free form statements, minimal interstatement semantics, no flow-of-control statements, comments and an editable[1] ASCII database.

Part of the basic **hobgoblin** concept is that it should only create output when an inconsistency is detected. A well written description checked against a conforming system should result in no output for the system manager to peruse.

The end result must be a tool that does not increase the amount of work that a system manager does yet extends the scope of his or her perception and influence.

Flexibility

We don't know beforehand all the types of consistency checks we want to use. Therefore, the **hobgoblin** language is designed to allow external, user defined checkers.

Performance Considerations

On the flip side, we hard-coded checkers in the interpreter to reduce the number of subprocesses to spawn.

[1] *ed*ible, *via*ble or *emac*ulate by hand.

The **hobgoblin** language also expresses the structure of directories in the filesystem. The interpreter uses this knowledge to increase performance by opening a directory just once when accessing the files listed as its contents, rather than once for each file.

Hobgoblin, The Language

Syntax and Semantics

A file description statement consists of a list of file names, an existence operator (':' in the illustration), a list of file attribute checkers, each with their associated argument (attribute) lists, and a statement terminator (';'). This list of file attribute checkers may include a directory contents checkers with nested description statements. Figure 1 shows the format. The are no interstatement semantics. A statement cannot even modify the existence operator or list of property checkers of a subsequent statement with the same file name. In other words, every statement stands on its own and is checked against the filesystem independently of any other statements. All interstatement semantics have been relegated to the **hobgoblin_delta** language and its interpreter.

The List of File Names

The list may contain any number of file names separated by white space. If the statement is nested inside a directory contents check, the file names inherit the path of the enclosing directory. Standard file name globbing characters as in Bourne shell and C shell may be used. File names containing tabs or spaces or characters considered special by the **hobgoblin** and **hobgoblin_delta** lexical analyzer will need to be quoted with single or double quotes. These special characters for **hobgoblin** are:

```
:  ?  !  [  ]  {  }  ;  #
```

The special characters for **hobgoblin_delta** include the above, plus:

```
+  -  =
```

which are only special characters as the first character in a statement. The special characters for file name globbing:

```
*  ?  [  ]  ^  $
```

also need special treatment. Since the globbing character set overlaps the other sets, you will often need to quote a globbed file name with *double* quotes to allow globbing but protect the special characters from interpretation by **hobgoblin**. Globbing characters can also be quoted into ordinariness with

```
file1 file2 ... fileN :
   checker1 [ attr1 ... attrN ] ...  checkerN [ attr1 ... attrN ]
   dir-content-chk { statement1 statement2 ... statementN } ;
```

Figure 1: The file description statement

single quotes or back-slash quoting. Using single quotes and back-slashes together currently does not work wholly as expected.

The Existence Operator

The existence operators define how **hobgoblin** treats the presence or absence of listed files. The existence operators are:

```
:   ?   !
```

Colon represents **must exist**, question-mark represents **may exist**, and bang represents **must not exist**. In nested descriptions, these also modify the otherwise inflexible **exclusive** and **inclusive** directory contents checkers.

The File Attribute Checkers

These checkers may be internal or external to the **hobgoblin** interpreter. If the checker is external the full or relative pathname to the checker program must be specified. The checker is followed by a list of file attributes enclosed in square-brackets. The checkers themselves are technically not part of the language and are arbitrarily named and dependent on the implementation. Nevertheless since they attain nearly the status and luster of reserved words once they are hardwired into the code, it seems to make sense to discuss them here.

Some of the more important internal checkers are:

- **mode** which checks the mode against regular expressions describing acceptable modes. We may someday extend **mode** to accept numeric modes such as those used with **chmod**.

```
phonelist responsibilities :
            mode[ -r..r..--- ];
```

- **user** which checks the owner of the file. This was named to conform with the naming of **find**'s related **-user** operator.

```
phonelist responsibilities :
            user[ root ];
```

- **group** checks the group ownership of the file.

```
phonelist responsibilities :
            group[ staff ];
```

  Because mode, user, and group are so commonly of interest and so commonly specifiable as an exact match on all three for many files, we have considered adding a **mug** check that would take exactly three arguments.

```
phonelist responsibilities :
   mug[ -r..r..--- root staff ];
```

- **size** checks the size. It accepts only a single parameter, either an integer, denoting exact match, or one of the [in]equality operators

```
==   !=   <>   <=   >=   <   >
```

  immediately followed by a size integer to denote a range of acceptable sizes. "==" is redundant, of course.

```
/.rhosts : size[ <=1 ];
/.rhosts : size[ 0 ];
```

- **symlink** checks if the file is a symbolic link pointing to one of the files listed in the checker's attribute list. For example, if I make a hopefully false statement like

```
/usr/ucb/vi : symlink[ /bin/rm
        usr/local/bin/emacs ];
```

  I really hope that **hobgoblin** tells me

```
0: symlink[/bin/rm /usr/local/bin/emacs] :
            Not a symlink : /usr/ucb/vi
```

- **mtime, ctime, atime** check the file times, using regular expressions. The date is rigidly formatted, for instance note that two spaces must precede the "7" in the example below. We want to someday allow more of a free format for the date, at least as far as the "white space" in it. The date must also be enclosed by double quotes so that hobgoblin takes it in as a single attribute.

```
hobgoblin : mtime[ "Aug 7 10:.. 1991" ];
```

  We plan to implement a parallel set of checkers called **mdays, cdays, adays** or the like, that will function like **find**'s **-mtime**, **-ctime** and **-atime** operators. The name conflict may confuse users, but we think it better to conflict with **find** than with the **stat** struct member names in this case.

- **exclusive** checks if its list of file names matches the list of files actually in the directory. This list is the attribute list of the exclusive checker, not the file name list at the start of the statement. Any files present that are not listed in the attribute list will result in an inconsistency message. For example, the following description checks the contents of a directory:

```
/var/spool/cron :
    exclusive[
        FIFO
        atjobs
        crontabs
        cron.deny
        queuedefs
    ];
```

  and because the list is incomplete, it provokes the following inconsistency messages out of hobgoblin on the system I am writing on:

```
0: Existence[!] : Exists : at.deny
0: Existence[!] : Exists : .proto
```

  In other words, these two files actually exist, but the **hobgoblin.conf** says they should not. It might be nice to have file name globbing in

these lists. Another one for the future. See directory contents checkers with nested descriptions for globbing.

The File Attributes

You can see from the above that multiple attributes are acceptable to all checkers except **size**. An implicit OR operates on a list of attributes. If any one attribute is correct then no inconsistency message is printed. Restated: the inconsistency message is printed only if no attributes agree with reality.

The Directory Contents Checkers with Nested Descriptions

Syntactically, directory contents checkers with nested descriptions occupy a place in a statement similar to that of the file attribute checkers. However, the "attribute list" of the nesting directory contents checkers is not made of attributes, but a collection of hobgoblin description statements enclosed between curly braces. The nested descriptions' file names inherit the path of the enclosing directory so they will not have full pathnames.

The only checkers implemented for the nested description level are **exclusive** and **inclusive**. **exclusive** checks if the list of files in the statement exactly matches the list of files in the physical directory. Any files present in the filesystem that are not listed in the statements will get an inconsistency message, as well as any file in the statement not present in the filesystem (see Figure 2). which makes hobgoblin produce the inconsistency messages seen in Figure 3. A statement with a **may-exist** operator (**?**), tells hobgoblin to accept that file's presence or absence in the directory without commenting on it.

The **inclusive** directory contents check is close to a directory contents *non*-check. It makes no special checks on the contents of the directory beyond what is required by the existence operators in the enclosed statements. The chief virtue of **inclusive** is that the files inside the curly braces inherit the path to the enclosing directory just as in the **exclusive** check.

The Statement Terminator

All statements must end with the semicolon, even those ending with a directory contents check containing nested descriptions. Placing a semicolon after a closing curly brace does go against the grain for your run-of-the-shop, line-a-minute C programmer.

Example Descriptions

For people wanting a bigger example, Figure 4 is a description of a root filesystem.

```
/var/spool/cron : exclusive {
        queuedefs FIFO atjobs crontabs home_boy_cron:
            user[root] group[staff];
        cron.deny ? user[root] group[staff];
};
```

Figure 2: Example of *exclusive* usage

```
2: Existence[:] : Absent : /var/spool/cron/home_boy_cron
4: Existence[!] : Exists : /var/spool/cron/at.deny
4: Existence[!] : Exists : /var/spool/cron/.proto
```

Figure 3: Inconsistency messages

```
/ : mode[rwxr-sr-x] user[root] group[wheel] inclusive {
    .cshrc : size[133];
    .login : size[62];
    .profile : size[87];
    .rhosts : size[<=1];
    .cshrc .login .profile .rhosts
        : mode[-r.-r--r--] user[root] group[staff];
    dev etc home mnt sbin var : mode[drwxr-sr-x] user[bin] group[staff];
    bin lib sys : mode[lrwxrwxrwx] user[root] group[staff];
    tmp : mode[drwxrwsrwt] user[bin] group[staff];
};
```

Figure 4: Description of a root filesystem

Writing Understandable Configuration Files

Here are some description formatting suggestions we would like you to consider when creating **hobgoblin** configuration files:

- Even if statements are short, at most have one to a line.
- Take the time to group files with identical attributes together in one statement.
- Because of the importance of the existence operators (**: ? !**) we recommend the practice of surrounding the existence operator in a statement with space rather than cramming it up against the preceding string. For instance compare (bad):

```
bin etc lib usr: user[root]
                 group[staff];
```

and (good):

```
bin etc lib usr : user[root]
                  group[staff];
```

We call it an operator, so think of it as syntactically similar to an "equals sign."

- It is more readable to place any nesting directory contents checker last in the list of checkers. Checker information about the directory will then remain near the name of the directory and not languish in isolation after the closing curly brace. This is only to keep related information close together. Compare (not so good):

```
/ : inclusive {
    .cshrc : size[133];
    .login : size[62];
} mode[rwxr-sr-x] user[root]
                  group[wheel];
```

and (better):

```
/ : mode[rwxr-sr-x] user[root]
        group[wheel] inclusive {
    .cshrc : size[133];
    .login : size[62];
};
```

The **hobgoblin** language does not put many restrictions on the format of a file description statement. We presume one can easily write inscrutable descriptions. Inscrutability naturally increases the time it takes to maintain description files and violates the principle that hobgoblin should reduce the system manager's work load. We hope the sensible **hobgoblin** user will apply the same rules of thumb for formatting descriptions that help maximize readability for any programming language source code.

Hobgoblin, The Interpreter

We chose to write the interpreter in C to try to provide both portability and execution efficiency. Efficiency is important because of the potentially immense size of the descriptions. Machine independence is important since we maintain systems from five different vendors.

What is Expected of a Checker?

Checkers are expected to do nothing if the properties of the file have a match among the checker arguments. Otherwise, the checker must print an inconsistency message of the form shown in Figure 5. on stdout. Some real live examples appear in Figure 6.

Internal versus External Checkers

Our basic guideline for whether to hard-code a checker in the interpreter hinges on whether the inode or directory contents contain the information. If the information we want to check is contained inside the file and it is a regular file, we leave it to external checkers. If the information of interest is in the inode or in the contents of a directory, it is nice to include checkers for that information inside the interpreter since inode and directory information is already gathered for every file that **hobgoblin** checks.

```
line_#: checker[attributes]: real_attribute: file_name
```

Figure 5: Checker inconsistency output

```
  20: mode[drwxrwsr-x] : drwxr-xr-x : /home
  33: mode[drwxr-sr-x] : drwxrwxr-x : /u
  41: mode[drwxrwsr-x] : drwxrwxr-x : /
 287: user[root] : cc : /etc/aliases,v
 320: mode[-r--r--r--] : -rw-rw-r-- : /etc/hosts.equiv
 348: user[root] : mil : /etc/rc.local
 365: mode[drwx--s--x] : drwx--x--x : /etc/security
 560: Existence[!] : Exists : /usr/mil
 592: Existence[:] : Absent : /var/spool/lpd
```

Figure 6: Sample output form the checker

We have been interested in including a **checksum** checker. This is an interesting problem because a single checksum algorithm will not agree with "sum" utilities on all systems, and besides may not be ideal for every purpose. If a built-in checksum does not agree with an easily usable checksum utility, then we would also need to provide this utility in some form for generating descriptions. Perhaps this would be best provided in skeleton form as an external checker. For instance, if you use a version of tar that will put out checksums of tarred files, you might wish to hack the checksum utility into **tar2hgl**. This would agree with the inode/directory-contents guideline for internal checkers. Then again, maybe we should pause here for some POSIX hand-waving.

Checker Interface

The interface to external checkers consists of **hobgoblin** providing the file name and the argument list from the description statement in the parameter list. Parameter passing for internal checkers is similar except it includes the *stat* file status structure with its data from the inode. External checkers are responsible for getting their own data about the file, including inode data.

Hobgoblin Cookbook

Setting Up

Starting up **hobgoblin** can really put a load on your disk and mind space because of the massive descriptions needed to describe a typical filesystem. Once you have a working **hobgoblin** executable compiled, you can get a starter description in three ways.

- You can write one by hand. This is fine for keeping track of small numbers of things, and ideal for close monitoring of a small software subsystem. For instance you might want to include in some script a bit like:

```
#!/bin/csh
hobgoblin << EOF_hgl
  /tmp : inclusive { * :
            size[ <600000 ]; };
EOF_hgl
```

  which is looking in /tmp making sure all files are less than 600000 bytes. Any file bigger than that will get a complaint from **hobgoblin**. Hand editing is also the only way right now to optimize a description for fast checking by grouping like files in a directory together in one statement.
- Alternatively, you can take a snapshot of your system with:

```
ls -Ralg / | ls2hgl > hobgoblin.conf
```

  The way we recommend to do it is to get **tar** listings from system and software package distribution tapes under SunOS 4.1.1 is shown in Figure 7, but this will vary wildly depending on your system.

Once you have your basic description, especially with the **tar2hgl** option, the hobgoblin will find enormous numbers of things to complain about when it starts comparing it to your system. Differences can be ironed out by hand. Many of the errors hobgoblin reports have to do with files installed in directories we would like to check **exclusive**ly, and are easily cured by adding statements about those files. We hope that eventually much of this kind of work will be done with **hobgoblin_delta**.

Dealing with Deltas (When They Come)

Local differences between the installation and distribution will need to be recorded as deltas. These deltas can be applied to the raw hobgoblin.conf off the distribution tapes with the code in Figure 8. **hobgoblin_delta** is the delta pre-processor, *hobgoblin_delta.confs* are presumed to be your files full of delta statements, *raw_hobgoblin.confs* are your raw hobgoblin descriptions straight from the distribution tapes,

```
#!/bin/csh
#  ...just as a frinstance...
mt -f /dev/nrst8 asf 5
foreach i ( root usr Kvm Install Networking )
  (dd if=/dev/nrst8 | uncompress | tar tvf - ) > sun4-4.1.1.${i}
  tar2hgl < sun4c-4.1.1.${i} >> hobgoblin.conf
  mt -f /dev/nrst8 fsf 1
end
```

Figure 7: Recommended snapshot algorithm

hobgoblin_delta *hobgoblin_delta.confs raw_hobgoblin.confs* > **hobgoblin.conf**

Figure 8: Dealing with deltas

and **hobgoblin.conf** is the finished, tailored hobgoblin.conf ready to install on the target system. This way you start from a well-defined position and system tailoring doodads that get reapplied at every OS upgrade are recorded in an organized fashion and can be verified easily. The raw hobgoblin.conf files from tape or **ls** should not be touched after creation.

Crontab

We use the cron entry shown in Figure 9. Again, **hobgoblin.conf** is how the description file is named on our all systems. **/usr/ucc/etc/hobgoblin.conf** on most of our systems is a link to **/var/ucc/etc/hobgoblin.conf** so that even systems mounting /usr from afar can have their individual **hobgoblin.conf**s. The "tricorder" account receiving the mailed output is our "On-Duty Sysadmin" who divvies up the sysadmin mail to whoever is around and can handle the problem described [1].

Maintenance

Once this is done for a machine you may still not be home free. If you wish **hobgoblin** to check things that are different on different machines, you will be tailoring your **hobgoblin.conf** to fit. Even if you have only one machine, you will still have to change **hobgoblin.conf** many times over the life of your installation. Each time you make a change to the master **hobgoblin.conf** or tailoring files, you have to remake and reinstall **hobgoblin.conf** for each of your machines. To keep this easy, you need to keep your source "hobgoblin_delta.conf" and "raw_hobgoblin.conf" types of files in an organized and accessible location, under RCS or SCCS control, with information about how to make new **hobgoblin.conf**s for each machine, and where and how to deliver and install them.

We have a mechanism that we use to take care of **make**ing and **rdist**ing, though we have not fully exercised its capabilities (read "tested it") and want to change some of the funny stuff in the original design before offering it to the public. It also depends on having a program called **suex**, that resembles a simple-minded version of the POSIX **tfadmin** command enabling system administrators to execute super-user commands from staff accounts. We need **suex** because we prefer to disallow root to **rdist**, yet staff cannot read, write, or run many system files. Getting a site to accept **suex** may take a massive leap of faith compared to trying out **hobgoblin**. It may be easier to write a big makefile and a big distfile and keep them up-to-date with each other by hand, just for starters.

Hobgoblin Deltas (Warning: Vaporware Alert!)

Motivation for Deltas

We want to automate tailoring of raw descriptions to fit particular machines. We want to express the differences between a description derived from a distribution tape and the reality of the filesystem in a compact form. These desires are satisfied in part by the **hobgoblin delta preprocessor**, otherwise known as **hobgoblin_delta**. We need nearly a whole 'nother Usenix paper to discuss it...

Delta Interpreter

The program **hobgoblin_delta** is written in perl. Since it only needs to run at description creation time, it does not need the lickety-split of C. Perl offers marvelous memory and data management, so it is ideal for an application that memorizes reams of data and churns it all around before spitting it out.

It reads both **hobgoblin_delta** delta statements and **hobgoblin** description statements, modifies the descriptions as directed by the deltas, and outputs a finished **hobgoblin** description minus deltas and comments. In other words, it is something of a compiler for **hobgoblin**.

Delta Language

hobgoblin_delta syntax is identical to **hobgoblin** syntax with the addition of the delta operator at the beginning of statement. The two character delta operator describes how the delta information is to be applied to a description. The first character of the delta operator designates whether the delta information will replace (=), get added to (+), or be deleted from (–) the hobgoblin description. The second character of the delta operator points at which syntactic elements will be manipulated: (**e**) for the existence check, (**c**) for the checker list, (**a**) for an attribute list, (**s**) for the whole statement, (**d**) for a directory contents check, and (**n**) for no part of the statement. A delta statement is only applied to statements for files listed in its own file list. For example the delta shown in Figure 10 says to remove the **mtime** checker from the **/etc** and **sendmail.cf** statements, remove the **size** checker from the

```
44 2 * * * /usr/ucc/etc/hobgoblin </usr/ucc/etc/hobgoblin.conf |
                /usr/ucb/mail -s hobgoblin-cpuname
                tricorder@cc.rochester.edu 1>/dev/null 2>&1
```

Figure 9: crontab entry

sendmail.cf statement, change **sendmail.cf**'s mode and **/etc**'s owner, and switch the directory check from **inclusive** to **exclusive**. Applied to the distribution tape description shown in Figure 11 should result in the result shown in Figure 12.

Current State and Future Directions

hobgoblin is checking root and /usr nightly on all of our Unix machines from cron. The same description is currently used everywhere.

Performance

Our implementation goals for the interpreter were: 10 file checks per second on a VAX 11/750 with a RA81 disk, BSD 4.n and System V source compatibility and machine independent source code. We met this timing goal easily even back when the interpreter treated the file system as a flat name space. We can no longer tell exactly how much we have improved over the original since the VAX is gone. The current hobgoblin interpreter running on a Sparcstation 1 can audit about 90 files and the contents of 20 directories per second with an average of 3.1 internal checkers per file description. These measurements were taken using a description from a tar file listing producing 42 error messages per second. This is a higher error rate than is desirable from a production **hobgoblin** description, but this rate of output barely changes the execution timing.

Utilities

Current

tar2hgl and **ls2hgl** generate **hobgoblin** descriptions. **ls2hgl** takes output from **ls -aglR** listings and writes flat descriptions where all file names are full path names. **tar2hgl** turns **tar tvf** listings into nice nested descriptions.

Future

Our first priority is to complete the **hobgoblin_delta** preprocessor.

The C macro and include pre-processor **cpp** restricts the use of the comment character (#) we want. This makes it unpleasant to work with for hobgoblinesque purposes. We want to beef up an existing preprocessor, **hpp** which will be added to the **hobgoblin** distribution, and use it to deal with macro expansion and include files. We want to pipe the following through the macro preprocessor:

```
# $SYSADMINS = dmo gort mil mking
# <private.conf
phonelist : user[$SYSADMINS];
```

and pretending that the file **private.conf** contains

```
emaillist faxlist snailmaillist :
                    user[$SYSADMINS];
```

get out the other end:

```
emaillist faxlist snailmaillist :
          user[dmo gort mil mking];
phonelist : user[dmo gort mil mking];
```

Macro and delta preprocessors will allow us to write more abstract (read "better") descriptions of systems.

Future Enhancements

We wish to make **hobgoblin** able to build itself on a wider variety of platforms. This is primarily a **make** problem.

```
-c /etc : mtime[ ];
=a /etc : user[root];
=d /etc : exclusive{ };
=a /etc/sendmail.cf : mode[-r.-r.-r--];
-c /etc/sendmail.cf : size[ ] mtime[ ];
```

Figure 10: A sample delta

```
/etc : mode[drwxr-sr-x] user[bin] group[staff]
     mtime["Oct 13 23:26 1990"] inclusive {
    sendmail.cf : mode[-rw-r--r--] user[root] group[staff]
      size[8334] mtime["Oct 13 22:39 1990"];
```

Figure 11: Distribution tape description

```
/etc : mode[drwxr-sr-x] user[root] group[staff] exclusive {
    sendmail.cf : mode[-r.-r.-r--] user[root] group[staff]
    };
```

Figure 12: Result of description in Figure 10

hobgoblin itself needs a few more internal checkers added and bugs subtracted as detailed previously.

The original spec calls for a **recursive** statement or directory check. This would allow describing something general in a rootward directory, say "No core files" ("core !;"), and have that check for that file name performed in that directory and all subdirectories below it in the directory tree. We are not even sure how that ought to be notated. Current thinking says leave it to **find**.

We have also played around with an external checker that could checksum selected lines of a file. Individual lines could have their own checksums and checksums could ignore white space and comments. The inconsistency message could include the text of the offending line.

On the more pie-in-the-sky plane we wonder if it might be nice to have *enforcers*, checkers that correct the properties of files when they do not match any checker arguments.

Summary

hobgoblin has fulfilled many of the expectations it was designed to meet. It now monitors the root and usr filesystems on all our machines. Eventually we want to audit all supported system software and configuration files on the machines we manage.

With **hobgoblin** in production, we are learning that we need ways to handle differences between related descriptions of filesystems and deal with the huge descriptions themselves more system-programmer-friendly fashion. This has stimulated the development of the **hobgoblin_delta** preprocessor and other utilities.

Availability

hobgoblin source and documentation is freely available via anonymous ftp at cc.rochester.edu in ftp/pub/ucc-src/hobgoblin. Or send email to problem@cc.rochester.edu if the nightly clean has gobbled it up. It has been compiled and run on Sun 3, Sun 4, Solbourne, Alliant running Concentrix, Vax running BSD, and Vax running Ultrix.

Author Information

Kenneth Rich has spent three years behind the scenes at the University of Rochester Computer Center working on distributed maintenance of ?*ix machines, departmental administrative programming, Ingres consulting, VMS networking and other projects. Before that, he slung diskpacks and Business Basic at a lumber yard. He also tutors at State University of New York, Empire State College. USMail: UCC - Towne House, 727 Elmwood Avenue, Rochester, NY 14620. EMail: (internet) kenr@cc.rochester.edu, (bitnet) krich@snyescva or kenr@uordbv.

Scott Leadley owns and runs Nanosecond Software Development producing printer software and other nifty things. He worked for the University of Rochester as a Systems Programmer/Systems Administrator in the ?*ix Operating Systems Group at the University Computing Center. USMail: Nanosecond Software Development, 1025 University Ave, Rochester, NY 14607. Fax and Voice: (716) 256-0370.

References

[1] Ondishko, D., "Administration of Departmental Machines by a Central Group", Proceedings of the Summer 1989 USENIX Conference, pp 73-82, 12-16 June 1989.

[2] Simonson, J. S., "System Usage Accounting at the University of Rochester", submitted to the USENIX LISA Conference, September, 1991.

Monitoring Activity on a Large Unix Network with perl and Syslogd

Carl Shipley & Chingyow Wang
- Jet Propulsion Laboratory, California Institute of Technology

ABSTRACT

This paper describes a set of simple perl scripts that can be configured to monitor system use on a large Unix network. This system is controlled by a main routine, meter, that coordinates initialization and scheduling of monitoring subroutines. The system allows one to develop short, highly modular tools to read Unix usage statistics, check them against configurable limits, relay them to remote System Administration workstations via *syslogd(8)*, and/or write them to a local log file.

Introduction

The Space Flight Operations Center (SFOC) development LAN at the Jet Propulsion Laboratory (JPL) consists of about 100 Unix workstations with over 220 currently active accounts. The LAN has machines on several different floors in four buildings and is used by personnel from a number of JPL projects. The System Administrators (SAs) might not normally go to the location of a machine in another building for days and might have limited personal knowledge of who the normal users of a machine are and what they do. All machines share a common password file via NIS or script-based remote copies and most user home directories are NFS mounted on all machines. A user can thus work on almost any computer on the the network. In this situation, it can be difficult to monitor use of system resources.

Unix provides a number of standard process accounting programs that can be used to monitor some aspects of system activity. However, in general, these programs, which provide data that is designed at least in part for use in billing users, do not provide the kind of information we want for monitoring system use. For example, the cumulative nature of the statistics kept by the accounting programs does not provide a good record of system use as a function of time. We want this kind of record so that we can answer questions like - who's been killing the system over the last half hour, or when is a good time to use a particular machine. In addition, the system accounting programs differ across the various platforms we have on our network, making it difficult to use them to develop a uniform monitoring scheme.

The kinds of tools we need to understand the daily activity of the system are basic Unix utilities like *ps*, *vmstat*, *df*, and *pstat*. However, Unix does not provide an easy way to combine these tools into a system for continuous monitoring of activity on a large network. One can write shell scripts to save information to a file for a specific machine, but only a few standard Unix commands, like *rup*((1),) are designed to be used in a networked environment.

We have developed a set of perl scripts that monitor aspects of system performance that are of interest to us and that provide a simple mechanism for distributing this information around a local network. These scripts record system usage information in a way that is flexible and can be easily tailored to specific situations. We used perl for this project because it is more powerful than the shells, easier to program, and has features such as built in functions for using *syslog(3)* routines that make it ideal for developing a system to monitor use of a large Unix network. The programs were developed and have been used primarily on BSD based systems, but should be portable to System V installations with a few modifications.

The Meter Program - Overview

Our system monitoring process is controlled by a perl script named meter. Meter itself is brief; its main role is to set up the monitoring environment and schedule calls to perl subroutines that perform various system measurement and data distribution tasks. The meter program initializes variables, opens a local log file named from the day of week (and changes this file if the day changes while the program is running), reads a configuration file that determines what subroutines will be used to monitor system use, and maintains a timing loop that computes the time to sleep between calls to system monitoring routines. Meter has 3 main sections:

1. initialization,
2. reading system statistics, and
3. writing statistics to either syslog or a local log.

The activities in 2 and 3 are repeated at a configurable interval in an infinite loop.

The program is controlled by a configuration file, meter.conf, which we keep in the same directory as the meter program. This directory is part of a

small file system containing various scripts used for system administration that is mounted on all of our machines. The conf file lists each machine on the LAN and identifies parameters and perl subroutines that are to be used to monitor activity on that machine. It is easiest to explain this configuration file with an example. Figure 1 presents a very simple set of configuration lines for a machine called wayback.

```
>wayback
+:DEFINE:$slpsec=300
-:INCLUDE:require
        '/u/scripts/diskless.conf'
+:READ:&rdup
+:WRITE:&syswrtup(10)
-:WRITE:&wrtup
```

Figure 1: Configuration for 'wayback'

In this list, the machine name is indicated by the "greater than" symbol in the first column. Configuration lines follow this name, up to the next line beginning with a "greater than" symbol. The initial '+' or '-' indicates whether a configuration line is to be used. This allows one to leave disabled configuration lines in a machine's conf file, to be used in a different situation (illustrated by disabled INCLUDE and WRITE lines in Figure 1).

Program Initialization

The meter program reads meter.conf each time it starts. In addition, it reinitializes itself on receiving a HUP signal, which it catches with the perl 'signal' call. To facilitate sending signals to the program, meter stores its pid in /etc/meter.pid on startup.

Individual entries within a meter.conf configuration line are delimited by colons, making it easy for the meter program to parse the line with the perl 'split' instruction. The second field of each line is a keyword. Each field that follows a keyword must be a legal perl command.

The program recognizes 4 keywords, DEFINE, INCLUDE, READ, and WRITE. The DEFINE keyword is used to initialize a variable or array of variables which are passed to the meter script and executed with the perl 'eval' command. In the example in Figure 1, $slpsec is initialized to 300. The $slpsec parameter is used to determine how long meter sleeps between successive system measurements. Its default is 600 but this can be overridden since DEFINE strings are evaluated after defaults have been set.

The INCLUDE keyword can be used to include a text file in the meter program using the perl 'require' command. This command functions much like a C include statement so that the contents of the required file become part of the meter code. It is thus possible to INCLUDE a file with all the desired contents of the meter.conf configuration list for a given machine, so that the only configuration line enabled is a single INCLUDE directive. This is a useful device for configuring classes of machines, like diskless nodes, that might run identical meter configurations. In the example in Figure 1, an INCLUDE statement of this kind is present but is disabled. The structure of a global configuration file INCLUDEed in this way must be slightly different from a simple list of configuration lines, since it has to provide the packed configuration arrays discussed below, rather than letting the meter program build them.

Subroutine calls are identified by the READ and WRITE keywords. These keywords correspond to labels within the meter program, the location of which controls the timing of the execution of subroutines read from the conf file. In the configuration line, the READ or WRITE keyword is followed by a field with a subroutine name prefixed with the & character, the standard perl syntax for a subroutine call. Like variable values, the subroutine calls are read into the meter program as simple strings which are executed with the 'eval' command. In the example in Figure 1, 'rdup', 'syswrtup', and 'wrtup' are perl subroutines that perform various monitoring functions dealing with the Unix *uptime* command. The meter routine reads subroutine names into @rd and @wrt arrays (the @ signifies a perl array variable). In the simple example in Figure 1, the rd and wrt arrays would have one only value, but in a real application the arrays would generally have several values. Inclusion in an array is determined by the keyword and order within the array is determined by the order lines are read in the conf file. The subroutines are executed in sequential sections of the meter script; all READ routines are executed before any WRITE routines.

Subroutine arguments can be passed directly with the subroutine names. These arguments can be either symbol names or as actual values. Variables are global by default in perl; their scope can be restricted by either the "local" or "package" instructions.

Reading System Statistics

In the example in figure 1, subroutine 'rdup' is a perl script that reads system load using the *uptime* command. This kind of subroutine could be a single line of perl, e.g.,

```
sub rdup { $rdupout=`uptime`;}
```

More commonly, the routine would also use perl regular expressions to extract desired information, which might be the number of users and/or the load over the last 5 minutes. The results would be packed into a form that could be easily parsed by WRITE routines that might use the data - e.g., with a brief

label and colon or comma delimiters between fields. A routine to produce this output would still be very short:

```
sub rdup {
 chop($tmp=`uptime`);
 $tmp =~ /^.*(\d+)
    user.*(\d+.\d+),
    (\d+.\d+), (\d+.\d+)$/;
 $rdupout = "up,$1,$3";
}
```

Packing information in this way is obviously important if it is to be saved to a disk with limited space. The output of a READ routine is saved in a variable that is typically named after the routine that read it, as in $rdupout. This variable must be a global, or declared as part of a package statement, so that it will be available to subsequent routines.

Checking Limits and Sending Messages via Syslogd

WRITE routines work with the output of READ routines. In the current implementation of meter, the name of the output buffer from a READ routine, and its structure (i.e. the order of variables in the output string and the delimiters) are hard coded into the WRITE routines that use this data. WRITE subroutines can be used to compare system statistics to limits specified as subroutine arguments in the meter.conf file. In the above example, syswrtup is a brief perl routine that compares a load value originally read by rdup to a limit and calls syslogd if the load is above that value. In this example, the value '10' in the syswrtup(10) call specifies the upper load limit to allow before using the syslog 'notice' call. The code to do this could be quite brief. Perl provides access to the syslog daemon through the 'openlog', 'syslog', and 'closelog' calls. The code for a very simple syswrtup subroutine, consistent with the examples used above, might be:

```
sub syswrtup {
  local($noticelim)=@_;
  ($lbl,$nusers,$ld5)=
          split(/,/,$rdupout);
  if ($ld5 > $noticelim) {
    &openlog('meter','','local0');
    &syslog('notice',$rdupout);
    &closelog();
```

The entry for the local /etc/syslog.conf file could be:

```
local0.notice   @sherman
```

where sherman is an SA machine. The entry in the /etc/syslog.conf file on sherman could be:

```
local0.notice    /var/log/daemonlog
local0.notice    /dev/console
```

which would send the $rdupout string to sherman's console and log. The log file entries would contain a time stamp, the hostname of the machine that sent the message, and the name of the meter program, as well as the $rdupout string, e.g.,

```
Jun 16 01:29:07 wayback meter: up,17,15.03
Jun 16 01:34:07 wayback meter: up,17,11.09
Jun 16 01:39:06 wayback meter: up,17,10.10
```

Different levels of system messages can be sent with syslogd. For example, a syslog notice message could be sent to the log file of an SA workstation for a load of 5, and a syslog alert message could be sent to the console for a load of 10.

Logging Information to a Local File

WRITE subroutines can also be used to log data taken from READ, or occasionally previous WRITE routines, to a local log file. The meter program takes care of the details of opening the log file and directing data to it by default. Thus, a typical WRITE subroutine of this type can be brief, consisting perhaps of a single print command, e.g.,

```
sub wrtup {print $rdupout;}
```

However, WRITE routines can also be quite complex. A good example is writing the output of the ps command. Depending on its options, ps on a busy system might have more than 100 lines of output. It generally doesn't make sense to save all this information to a log file. It also might not make much sense to have the READ routine condense the output, since different WRITE routines might be developed to archive different subsets of a ps output, depending on the situation.

Example Applications

Using the above system, it is easy to develop customized monitoring routines that continuously log system statistics and/or notify System Administrators when various events occur. Having the number of users and load average for each machine on a LAN sent to a central SA workstation every 10 minutes is a simple but useful example. It is worth noting how easy it is to implement this with the meter program; all the code necessary for this application has been presented in the examples above, as well as the necessary syslog.conf statements. A more realistic example would be to make several system measurements and pack them into a single line sent to an SA machine's syslog file at some regular interval. For example, system idle, free memory, and page outs might be selected from the output of *vmstat*, swap and number of processes could be monitored with *pstat*, etc. Depending on how these statistics are packed, it is fairly easy to get 5 to 10 different system measurements on on a single summary line, including the syslog timestamp and host information.

This kind of sample, taken even at 30 minute intervals, would provide a reasonable overview of system activity across the network without producing a huge log file. Logging on the order of 80

characters each 30 minutes for each of 100 machines over 24 hours would involve a file of less than 400 Kbytes. This number could be reduced substantially by specifying some low load limit in a WRITE routine so that statistics would only be sent when the system had some minimal level of activity - thereby avoiding most reporting when the system is idle.

More specific metering routines can be developed for specific situations. For example, we occasionally suspect that a program (or person) is causing a problem on a machine. It is a simple task to write a perl routine that does a *ps* and a *grep* to determine if a specific process is running and notify the SAs via *syslogd*. Another common situation is that a developer or tester has a large program that he knows is crashing the system, but the problem seems to occur only sporadically. By running meter, we can keep a time-tagged record of system state as the program is running and examine the local log file to see what was going on just before the crash.

On our major servers, we keep more detailed system statistics than those in the examples above, and we respond to out-of-limits events by launching short programs designed to determine what (who) caused the condition. A more-or-less standard configuration for our severs keeps the following records:

1. Averaged results from three *vmstat*s separated by 10 seconds: statistics are archived for processes in the run queue, processes blocked for resources, processes runnable but swapped, available memory, page outs, percent cpu for user processes, system time, and percent idle. The first *vmstat* line, which reflects averages since system boot, is discarded.
2. Results from *df*s for local user disks, if they exist: percent free and blocks free. We record disk usage each hour and update it every 10 minutes if it changes.
3. Results from *ps*: a complete archive of users saved once an hour with cpu, mem, siz, and rss, and a list of processes being run by that user. This list is updated each 10 minutes for new processes and logouts. We do not list system daemons, but we print a count. For multiple instances of a process, e.g., several shells, the program prints a count in parentheses.
4. Results from *pstat*: number of processes and swap usage.
5. Results from *uptime*: number of users and load average over the last 5 minutes.

Our standard server configuration checks two out-of-limits conditions, disk usage (if the machine has local disks with user files - we do not check system disks) and load average. Limits for these values are read from the conf file. If one of the limits for disk usage or load average is exceeded during any 10 minute check, the WRITE routine launches a program designed to identify the offending user.

An important part of monitoring our network is keeping track of disk usage. We do not use disk quotas; instead, we allocate large disk partitions to groups of programmers and let the group supervisor determine who gets what. This provides flexibility to the various development teams, but requires that we monitor disk usage fairly closely because the system allows users to fill their group's disk entirely. As part of our approach to monitoring disk usage, we run nightly *dus* on all the user directories. The *du* records are saved to a file given the name of the disk partition and the day of the week. The program renames the old files by appending 'bak' to the file for that day, thereby erasing the existing 'bak' file. Thus, records are kept for two weeks. The perl routine that runs the nightly *dus* makes three lists: a simple list of the *du* results, a list of differences from results of the previous day and a list of differences from the previous week. These lists are sorted by size. This list gives the SAs a quick overview of which users are consuming the most disk space. If a disk is filled during the day, the standard *df* WRITE routine runs a program called *newdu*, that does another *du -s* on the user partitions for that disk, checks the results against the log file from the nightly *du*, and writes out a sorted list of changes. A glance at this list reveals who is filling up the disk.

Responding to high load averages is considerably simpler than the system for disk overruns. When the load gets above the limit set in the WRITE routine for uptime, it launches a perl script called 'topps'. This routine is similar to the C *top* program [2] but uses a perl sort of the %cpu field from a *ps -auxcc* to determine the top 10 users of cpu time. Topps writes this information to a log file every 2 minutes for 10 minutes and then exits. By reading the log file, an SA can usually determine who is driving up the system load, if the high load is due to a process using too much cpu.

Summarizing System Statistics

The output of these metering programs is extensive. Generally we can review it by scanning a log file, but it is important to be able to condense some of this output into a format suitable for a report that can be shown to users or managers. We have developed simple perl scripts to summarize data from meter logs in graphical form.

We use *giraphe3*, a public domain graphing package to generate on-line graphs of system use [3]. *Giraphe3* understands X11, and can draw a detailed graph of, for example, system load average over a week in an X window. *Giraphe3* can also output postscript files to the printer. We have developed a simple perl script that reads meter log files, creates the configuration and data files required by *giraphe3*,

and launches *giraphe3* to output the graph.

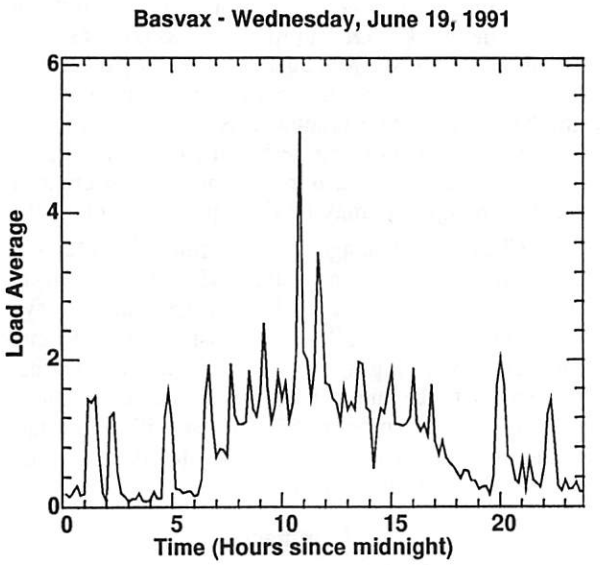


Figure 2a: Basvax's load

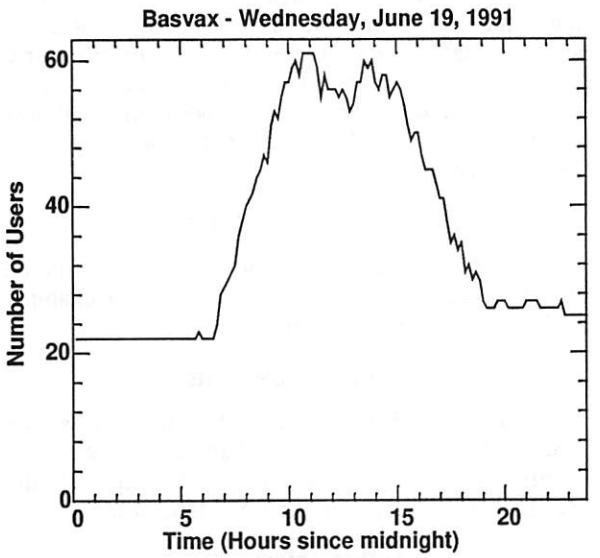


Figure 2b: Basvax's users

Figure 2 shows a typical report produced by *girahpe3* for number of users and load average on one of our busy machines over a typical day. An interesting feature of this graph is that a large number of users are logged into the machine even in the middle of the night. The computer from which these statistics were taken is a large VAX used primarily for news and mail. Many people keep a window open on it continually and some leave a mail reader running continuously. The machine does a does a good job of handling a fairly large number of users, each of whom is generally reading or writing text files. There is a slight upward trend in the number of users. This is a real effect that tends to build during the week and to be reset on weekends and holidays.

Figure 3 shows a more typical pattern of use and load, taken from one of our Sun4s used for program development. Compared to the VAX in Figure 2, this machine has many fewer users but a higher average load during normal working hours. In contrast to the VAX, when people are not using this machine they generally log off.

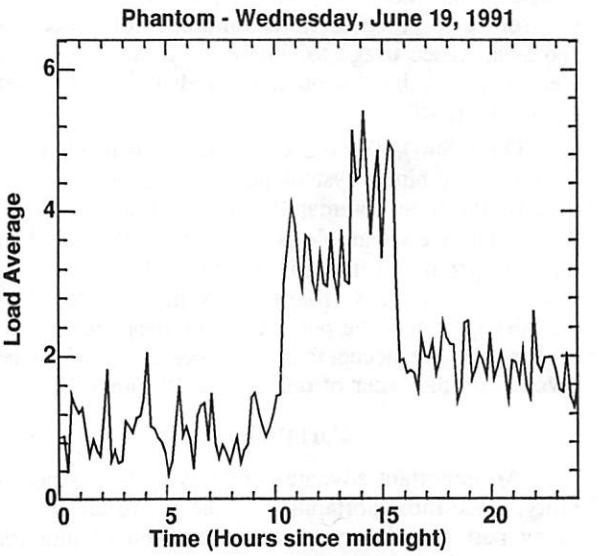


Figure 3a: Phatom's load

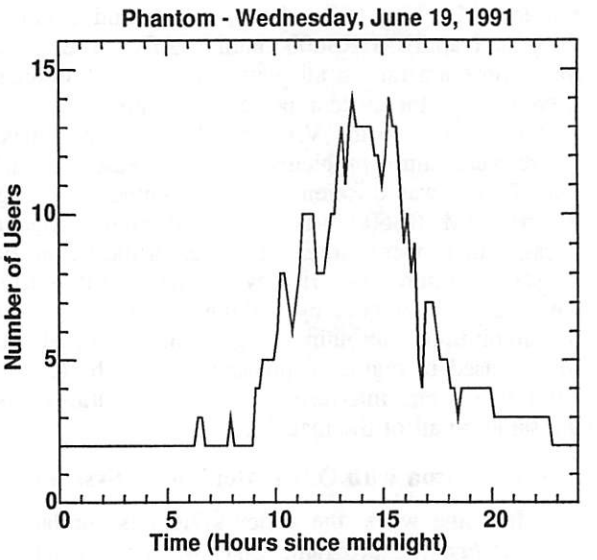


Figure 3b: Phantom's users

Assessing the Speed of perl Based Monitoring

In order to get a sense of how much cpu time we would gain by developing dedicated C routines to handle some of our monitoring tasks, we developed a C program to read statistics similar to those we might read in a standard meter application.

The C program used the Sun kvm library routines to examine the process table in /dev/kmem and the user structure in /dev/mem. We compared the time it took to run with that of a perl routine that called system utilities to read similar information. The C program simulated the output of a *vmstat* by monitoring the number of processes in the run queue, the number of processes blocked, the free memory, percent cpu use for system, user, and idle. It did a simulated *ps -auxcc*, reporting %cpu, mem, rss, and siz, for each process. It read number of processes and swap space usage to simulate a *pstat -T*. And it measured free blocks on a local disk in a manner analogous to *df*.

On a Sun4/490, the C program ran in about .2 seconds (combined system and user times as measured by the time command). It was at least 7 times faster than a comparable perl routine calling system utilities (from 1.3 to 1.6 seconds). While, as one would expect, the C routine was much faster than the perl approach, the perl-based routines executed at speeds that are acceptable for tasks designed to be invoked on the order of once every 10 minutes.

Portability

An important advantage of this system is portability, since most portability problems are taken care of by perl and the system utilities used to monitor activity. The meter program and our standard set of monitoring subroutines were developed on Sun workstations. We have ported them to a VAX 8600 running Mt. Xinu BSD Unix, a Pyramid running OSx, and an IBM RS600 running AIX. The basic meter program ran on all platforms. The subroutines used to monitor system performance ported almost without change to the VAX and Pyramid machines. There were minor problems caused because the output of *pstat* was different on the machines. Porting to the IBM RS600 was somewhat more difficult because there were more differences in the output of standard commands. However, these differences were easily overcome by making minor changes in the monitoring subroutines - generally to adjust the format used in regular expressions. The basic perl environment and interface to the syslog routines was the same on all of the machines.

Comparison with Other Monitoring Systems

In some ways, the meter system is similar to the *watcher* [4] program. Like *watcher*, meter is designed to optimize use of standard Unix utilities for monitoring system activity. *Watcher* also has provisions for checking limits. An obvious difference, however, is that *watcher* has no feature like the use of the *syslogd* routines which allow the meter program to distribute information across a large network.

While this paper was being written, we looked at a demonstration version of Sun Net Manager, which can also be used to monitor system activity on a large network. Our initial impression was very favorable. Sun Net Manager can clearly do most of the things that the meter program was designed to do and has an attractive graphical user interface into the bargain. It appears to be designed to be configurable so that the user can customize a monitoring environment, although this may involve quite a bit of work.

Clear disadvantages of the Sun Net Manager are that it costs money and that, at least in its present form, its use may be restricted to Sun systems. In addition, while it is possible to write customized monitoring agents for use with the product, it appears to be much harder to write these agents (which are C programs with specific interface requirements) than it would be to develop perl scripts to do similar things.

Summary

The meter system described in this paper uses simple perl scripts to monitor activity on a Unix network and the syslog daemon to automatically distribute this information to a central site. Meter is small, portable, easy to use, and easy to customize to individual situations. It can be configured to provide detailed data about system behavior in almost all the areas with which a System Administrator is generally concerned.

Program Availability

Please contact the authors for information on obtaining copies of the meter program and example perl scripts for use in system monitoring.

Acknowledgements

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Author Information

Carl Shipley received a Ph.D. in Experimental Psychology in 1981 from UCLA where his research interests focused on the use of computers to extract information from biomedical signals such as EEG. He has worked as a software engineer at JPL since 1982 and is currently the Technical Group Leader for System Administration and Software Environment in the Space Flight Operations Center development group. He can be contacted at carl@jpl-devvax.jpl.nasa.gov, or at the Jet Propulsion Laboratory, M/S 301-260, 4800 Oak Grove Blvd., Pasadena, Ca. 91109

Chingyow Wang graduated from National Central University at Taiwan at 1979 with a B.S. in Physics. He recieved an M.S. in Computer Science

from California State University at Fullerton in 1983. He has worked in Unix software development projects for the last 9 years and has been at JPL since 1987. He is currently a System Administrator and C consultant in the Space Flight Operations Center development group. He can be contacted at wang@jpl-devvax.jpl.nasa.gov, or at the Jet Propulsion Laboratory, M/S 301-260, 4800 Oak Grove Blvd., Pasadena, Ca. 91109

References

[1] Programming perl, by Larry Wall and Randal L. Schwartz, O'Reilly and Associates: Sebastopol, CA. 1991.

[2] Top is a program that identifies users with the highest cpu usage. It was written by William LeFebvre, Department of Computer Science, Rice University, phil@rice.edu

[3] Giraphe3 was written by Rober M. Harris (harris@caf.mit.edu) and Duane S. Boning (boning@caf.mit.edu).

[4] Unix Tool Building, by Kenneth Ingham. Academic Press: New York, 1991.

SCRAPE (System Configuration, Resource And Process Exception) Monitor

Richard W. Kint - College of Engineering, University of Washington

ABSTRACT

In system management, as in business management, dealing effectively with exceptional situations is one key to success. The **SCRAPE Monitor** allows system managers to define system models and describe systems in high-level terms. Systems are compared to these models to find file and process configuration exceptions so that potential problems may be fixed while they are (hopefully) still minor.

Introduction

In prehistoric times, computing sites had a small number of systems serving a large number of users; it was realistic to spend substantial amounts of time monitoring process configurations or checking file configurations in preparation for upgrades. Nowadays many installations have a large number of systems serving a large number of users, without a proportional increase in the number of administrators. It is simply not feasible to administer workstations individually, even if they are as complex as the minicomputers and superminis of the past.

Any business management text will say that a key to concise and effective reports is focusing on exceptions. This is also a key to effective system management, especially in large networks: the routine can often be ignored, but it is necessary to zero in on exceptional conditions quickly. Being able to find and fix little problems before they become big problems is a fundamental requirement for scaling up administrator-to-system fanout.

Configuration Consistency

One part of the solution is configuration consistency. To the extent that systems can be treated interchangeably, the amount of administrative overhead per system is reduced. Major changes can be applied globally, making them a single replicated operation rather than a series of separate operations. If configurations are rigidly controlled, exceptions are trivial to isolate.

Strict configuration consistency doesn't seem to last, however. Some of this is just entropy, but there are often legitimate reasons why complete configuration consistency is not always attainable or desirable: software may only be licensed for a particular host, or it may not be desirable to offer an expensive service on all hosts.

Since complete consistency is not always attainable, a system for monitoring configuration consistency must provide flexibility in describing configurations. It should bring unexpected exceptions to the attention of system managers while filtering out the expected.

The SCRAPE Monitor Toolset

The **SCRAPE Monitor** meets this goal. It allows a system manager to define configurations using reasonable building blocks such as the base O/S and major layered products. These building blocks (or *models*) can be combined, allowing divergent systems to be defined in high-level terms. A model (or group of models) can be checked against a system to generate an exception report. Models can also be excluded, allowing known exceptions to be ignored. To facilitate consistency, some attributes can be set from a model.

Currently two types of entities may be modeled, files and processes. The models are ASCII files containing objects and attributes. The **SCRAPE Monitor** consists of two complementary toolsets, one for files and one for processes, and a front end which calls these tools. The tools use the cryptic UNIX command syntax we all know and love, while the front end compromises with the user to the extent of using complete words. File models, the file tools and the front end will be discussed in detail and then the process tools will be briefly mentioned.

File Model

The file model format is based on **ls -l** and has the following fields:

mode The first character indicates the file type:

-	regular file
d	directory
l	link
c	character special
b	block special
p	pipe (FIFO)
s	socket

The next 9 characters indicate permissions and setuid, setgid, and sticky bits.

checksum A checksum of the file contents (regular files only).

blocks The size in blocks.

links The number of hard links to the file.

owner The name of the file owner.

group The name of the file group.

size The size in bytes. For special files, this field contains the major and minor device numbers.

mod time The time the file was last modified.

name The pathname of the file. This may be relative or absolute.

resolution For symbolic links, the link text is printed following an arrow (->).

A header contains a model name and version and the operating system and platform to which the model applies. An optional trailer (omitted to reduce clutter) gives the start and stop time for model generation and summarizes the contents of the model (total number of files and blocks, counts of files, directories, etc).

Sample File Models

Figures 1 through 3 are excerpts of file models. Figure 1 is part of a base operating system, Figure 2 lists C compiler components shipped with the O/S and Figure 3 is a model of an updated C compiler.

As a file model is a text file, some information is lost; only the modification date is stored, and it is impossible to tell if the world-execute bit is set if the sticky bit is set. However, the advantages of an easily understandable and editable format outweigh these disadvantages in practice.

fmgen

Fmgen creates a file model. Pathnames may be specified on the command line or may be fed through the standard output to allow sophisticated pipelines or files as a source of pathnames. Header information may be specified on the command line. A man page is provided as an appendix for those desiring more information.

```
#Model Name: Base O/S
#Model Operating System: HP-UX 7.0
#Model Platform: HP 9000/3XX
-r-xr-xr-x  D3547D10   71  1 bin  bin  71748 Oct 12 07:00 1989  /bin/ar
-r-xr-xr-x  11D08BC9   20  1 bin  bin  19588 Oct 12 07:00 1989  /bin/as
-r-xr-xr-x  350C8809   34  2 bin  bin  34348 Oct 12 07:00 1989  /bin/cc
-r-xr-xr-x  2AE4F8C2   26  1 bin  bin  26440 Jan 16 08:00 1990  /bin/chmod
-r-xr-xr-x  AC4E4F44  184  1 bin  bin 180224 Oct 12 07:00 1989  /bin/csh
-r-xr-xr-x  EB141AEF   50  1 bin  bin  50184 Jan 16 08:00 1990  /bin/grep
-r-xr-xr-x  02B20EA9  176  2 bin  bin 172032 Jan 16 08:00 1990  /bin/ksh
-r-xr-xr-x  81883696   59  1 bin  bin  60085 Oct 12 07:00 1989  /bin/ld
```

Figure 1: Base Operating System Model (excerpt)

```
#Model Name: C Compiler
#Model Version: 7.0
#Model Operating System: HP-UX 7.0
#Model Platform: HP 9000/3XX
-r-xr-xr-x  11D08BC9   20  1 bin  bin  19588 Oct 12 07:00 1989  /bin/as
-r-xr-xr-x  350C8809   34  2 bin  bin  34348 Oct 12 07:00 1989  /bin/cc
-r-xr-xr-x  81883696   59  1 bin  bin  60085 Oct 12 07:00 1989  /bin/ld
```

Figure 2: Old C Compiler Model (excerpt)

```
#Model Name: C Compiler
#Model Version: 7.4
#Model Operating System: HP-UX 7.0
#Model Platform: HP 9000/3XX
-r-xr-xr-x  838BB1FA  216  2 bin  bin 208896 Oct  2 15:27 1990  /bin/as
-r-xr-xr-x  6A172110   52  1 bin  bin  52572 Oct  2 15:27 1990  /bin/cc
-r-xr-xr-x  465B6582   92  1 bin  bin  94208 Oct  2 15:27 1990  /bin/ld
```

Figure 3: Updated C Compiler Model (excerpt)

The three models shown in Figures 1 through 3 are stored as separate files. Assume that there is a root for file models. This has a subdirectory for HP 9000/3XX systems running HP-UX 7.0. The first model would be *hp300_7.0/base*, the second would be *hp300_7.0/c_7.0*, and the third would be *hp300_7.0/c_7.4*. These are the building blocks used in further comparisons.

Typically, *fmgen* is used to generate much larger file models for basic system building blocks such as the base operating system, major options, layered products, and third-party products.

fmas

Fmas takes models produced by *fmgen* and assembles them into larger models. Models may be included or excluded. Differences in attributes between files of the same name in different models are reported as in *fmcmp* (see below).

Log files were a primary motivation behind the *exclude* option. They are frequently modified and will always differ from a model. Being able to exclude known (and unimportant) discrepancies facilitates concentration on real exceptions.

Using the models in Figures 1 through 3, the first model (the base O/S) is included as the basis. The second model (the original C compiler) is excluded; these files are removed from the combined model, and then the third model (the updated C compiler) is included. This is done with the command:

```
fmas -i hp300_7.0/base  \
     -x hp300_7.0/c_7.0 \
     -i hp300_7.0/c_7.4
```

Explicitly excluding the original C compiler may seem like overkill, but is consistent with the motivation behind **SCRAPE**. If the original compiler were not excluded, messages about the discrepancies would be printed whenever the models are combined. These are meaningless (since these files are known to differ) and only serve to divert attention from potentially significant exceptions. **SCRAPE** is designed to filter out noise, not to add more.

fmcmp

Fmcmp compares a file model to a system. Any of the attributes represented by fields in the file model may be selected for comparison; the default is to check for existence (treated as an attribute internally). Any of the file types may be selected as well, so that comparisons may be limited to (for example) regular files and links. A destination directory may be specified with pathnames in the model being interpreted relative to this. Normally only objects in the model are checked, but an *exclusive* flag causes objects not in the model to be flagged.

Fmcmp has two output formats, one designed for readability (to be sent to humans) and one designed for parsing (to be used in pipes). Assuming that the output from the *fmas* example above were sent to the command *fmcp -a ms* (selecting attributes modification date and size for comparison) and the target system still had the original C compiler installed, the output of *fmcmp* would be seen in Figure 4.

If this command were run after a global compiler update with the results above, it would be readily apparent that this host had somehow been skipped. An alert administrator could remedy this before users started screaming for blood.

fmset

Fmset allows certain file attributes to be set from a model. These include the permissions, the owner and group, the modification time, and the link resolution. Modifying contents is beyond the scope of this toolset, so attributes depending on contents cannot be set.

scrape

Scrape (the program, not the system) provides a front end to the last three of the file model tools, it is not used with *fmgen*. A *scrape* command file encapsulating the operations above is shown in

```
/bin/as:
    blocks:   model has 216, host has 20
    size:     model has 208896, host has 19588
    mod_time: model has Oct 2 15:27 1990, host has Oct 12 07:00 1989
/bin/cc:
    blocks:   model has 52, host has 34
    size:     model has 52572, host has 34348
    mod_time: model has Oct 2 15:27 1990, host has Oct 12 07:00 1989
/bin/ld:
    blocks:   model has 92, host has 59
    size:     model has 94208, host has 60085
    mod_time: model has Oct 2 15:27 1990, host has Oct 12 07:00 1989
```

Figure 4: fmcmp output

Figure 5.

```
model-type file
include hp300_7.0/base
exclude hp300_7.0/c_7.0
include hp300_7.0/c_7.4
type file link
check size modtime checksum resolution
format pretty
notify kint
```

Figure 5: scrape command file

On reading this file, *scrape* calls *fmas* and *fmcmp* with the appropriate options, and mails the output to the designated victim.

Scrape command files that only have *include* and *exclude* directives may be treated as models and included or excluded in turn. This allows standard configurations to be easily specified.

scraped

Scrape may be invoked with a hostname as an option. In this case model assembly takes place on the local system, but then the assembled model and associated *fmcmp* directives are shipped to *scraped*, a daemon running on the remote host. This allows polling from a central site. In such a case the central site maintains a model file (containing host-specific exceptions) for each host it polls.

Process Models

After using the file tools, it became apparent that the concepts could be generalized to cover processes as well. The existing code was modified to produce a related set of process tools. *Pmgen* generates a process model, *pmas* assembles process models, *pmcmp* compares process models to a running system, and *pmset* sets process attributes.

The format for a process is model is similar to that of the BSD **ps u**, with the entire command line being printed to allow a process to be restarted correctly. The principles are the same, but the attributes are different. In general the only attribute checked or set is existence. However, if a server is running with the wrong UID, *scrape* will catch it.

Design Principles

A guiding principle was **KISS**: Keep it Simple, Stupid. Familiar data formats were used to reduce the learning curve. An effort was made to write general, reusable code. The process model toolset was added some time after the file model tools, but was an easy adaptation.

The more complicated something is, the more likely it will break. So the **SCRAPE Monitor** was implemented using the lowest-level building blocks possible. It executes no system commands and uses a socket-level client and server. Troubleshooting tools need to work when the environment is hosed. This ruled out use of RPC, since at the time this was implemented (1989) RPC systems were often at the root of network problems.

Results

The **SCRAPE Monitor** performs its modest job well. The file model tools are very useful in tracking *ad hoc* configuration changes. It's easy to make a minor change to a host while putting out a fire, forget about the change as other fires start, and then to be burned by the change much later. But this system doesn't let you forget.

It turns out that the discipline of defining models and finding discrepancies is useful in itself. Putting together models for various products and observing discrepancies helps anticipate problems, and recording unavoidable configuration discrepancies makes it easier to plan around them.

Future Work

These tools were designed to make a system administrator's life easier. Only features with a clear near-term payback were implemented. Some that weren't:

In a heterogeneous environment, different sets of models are used for different platforms; the model facility is not generalized so that a generic model name may be used and then the version applicable to a given platform and operating system version fetched. Pathnames for models are chosen somewhat arbitrarily.

All the attributes and the checks performed thereon are hardcoded. It would be nice to make this more like a language in which a generic operation could be mapped onto a system-specific or object-specific implementation.

Scrape needs hooks into software installation and deinstallation tools, so that changes made to a host are tracked in its model configuration files automatically. Making these changes manually is not burdensome, but often not done before the next exception report. This demonstrates a simple fact of human nature: as long as system administration tools require extra effort that gets in the way of putting out fires, they will not be used all the time.

Availability

Source code for the **SCRAPE Monitor** is available via anonymous FTP from the host *ftphost.engr.washington.edu* in the directory */pub/local/scrape*.

Author Information

Rick Kint is a Software Engineer in the College of Engineering at the University of Washington, Seattle. Although happy with the major decrease in his blood pressure since leaving the ranks of system

administrators, he retains an active interest in system and network administration tools. He may be reached at the College of Engineering, FC-06; University of Washington; Seattle WA 98195 or through e-mail as kint@engr.washington.edu.

NAME

fmcmp - compare file model to directory tree

USAGE

fmcmp [*options*] [- / *modelnames*]

DESCRIPTION

For each *modelname* argument, **fmcmp** compares the items listed in the model to the actual filesystem objects of the same name. Output is not sorted; filenames appear in the order in which they appear within the model, and models are processed in the order in which they appear on the command line.

OPTIONS

- Read the model to be checked from the standard input.

-a *spec* (attribute) **spec** is a specification of the attributes to be compared. These may be chosen from the list

e existence
t type
p permissions
c checksum
b block count
l link count
u owner
g group
s size
d device major and minor numbers
m modification time
n name
r resolution

The special specification *all* selects all of these. The default is to check for existence.

-c *name* (change) Change directory to **name** before running comparison. This is useful for using models that use relative pathnames. This option may only be used once on a command line.

-t *spec* (type) **spec** is a specification of the file types to be compared. These may be chosen from the list

b block special
c character special
d directory
f file
l link
p pipe
s socket

The special specification *all* selects all of these. The default is to check all types.

-x (exclusive) Print files which exist on the target system but not in the model. The default is to only check files which appear in the model.

-n (name) Print only the names of files which differ in the selected fields.

-p (pretty) Select the user-oriented output. This is the default if the standard output is a tty.

-P (not pretty) Select the command-oriented output. This is the default if the standard output is not a tty.

OUTPUT FORMATS

There are two basic output formats, one designed for readability and one designed for parsing.

In the human-readable format, the filename is printed once for each file in which differences are found. Then for each differing attribute, an indented message is printed. The first field in this message is the attribute name, and the second contains the differences.

```
types.h:
        mod_time:       model has May 18 16:46 1989, file has May 30 17:57 1989
types.o:
        checksum:       model has 6BA29E83, file has 0F455DAE
        blocks:         model has 12, file has 13
        size:           model has 11772, file has 12776
        mod_time:       model has May 18 16:47 1989, file has May 30 22:30 1989
```

The second output format is designed to be parsed easily for use in pipelines. Each line has three fields, separated by tabs. The first is the filename, the second is the field in which the difference was found, and the third is the error message (which is always of the format manfest has X, file has Y).

```
types.h: mod_time:      model has May 18 16:46 1989, file has May 30 17:57 1989
types.o: checksum:      model has 6BA29E83, file has 0F455DAE
types.o: blocks:  model has 12, file has 13
types.o: size:    model has 11772, file has 12776
types.o: mod_time:      model has May 18 16:47 1989, file has May 30 22:30 1989
```

CAUTIONS

Newline and tab are considered printing characters in filenames.

FILES

/etc/passwd
to get user IDs.

/etc/group
to get group IDs.

AUTHOR

Rick Kint

SEE ALSO

fmas(1L), fmgen(1L), fmset(1L), model(1L), scrape(1L)

NAME

fmgen - generate file model

USAGE

fmgen [*options*] [- / *pathnames*]

DESCRIPTION

For each *pathname* argument, **fmgen** prints a model record. If an argument is a directory, by default **fmgen** prints a model record for the directory and each item therein. When no argument is given, **fmgen** prints model records for the contents of the current directory but not the directory itself. Output is sorted alphabetically for each argument, but not across all arguments. The model format is documented in **model(1L)**.

OPTIONS

-	Accept *pathname* arguments from the standard input (useful in using **find(1)** or some other command to generate a list of filenames).
-c	(comment) The user may enter comment text before model generation. End with EOT (usually ^D) or a line containing only a period ('.').
-d	(directory) If argument is a directory, generate a model record for the directory itself, not its contents.
-L	(link) If argument is a symbolic link, generate model information about the object the link references (the resolution) rather than the link itself.
-R	(recursive) Recursively process all subdirectories encountered.
-T types	(type) Only process files of the given type(s), as in **ls(1)**: **f** = regular file, **d** = directory, **l** = symbolic link, **c** = character special, **b** = block special, **s** = socket, and **p** = named pipe (FIFO).
-f	(fast) do not determine checksum or text type for files.
-h	(header) Print a header containing the time the model was started, the directory from which it was created, and the person creating it.
-n *name*	(name) Supply a model name which will be added to the header. This option turns on **-h**.
-o *opsys*	(operating system) Supply an operating system name which will be added to the header. This option turns on **-h**.
-p *platform*	(platform) Supply a platform name which will be added to the header. This option turns on **-h**.
-v *version*	(version) Supply a model version which will be added to the header. This option turns on **-h**.
-t	(trailer) Print a trailer containing the time the model was finished, a count of items (broken down by type), and a total of blocks and bytes.
-D	(debug) Print debugging output.

CAUTIONS

There is no special processing for non-alphanumeric characters in filenames.

The listing is quite wide.

Hard links receive no special treatment; **fmgen** makes no attempt to find out about other entries for files with a link count greater than 1.

FILES

/etc/passwd
to get user names.

/etc/group
to get group names.

AUTHOR

Rick Kint

SEE ALSO

fmas(1L), fmcmp(1L), fmset(1L), model(1L), scrape(1L)

Packet Filtering in an IP Router

Bruce Corbridge, Robert Henig, Charles Slater - Telebit Corporation

ABSTRACT

By using existing information in packet headers, routers can provide system administrators a facility to manage network connections between computers. Host address, network number, interface, direction, protocol, and port number are parameters that may be used to implement an access control policy.

We present experiences developing the packet filtering facility in the *NetBlazer* dial-up IP router. We address the sometimes conflicting design goals of efficient performance and ease of administration by choosing internal data structures that simplify per packet lookup and then devoting 90 per cent of our code to implementing commands that maintain these tables in manner that is easy for system administrators.

Introduction

Wide area networks provide remote sites convenient access to local networks. With this increased convenience comes the often complex problem of unauthorized access to network resources. Packet filtering in an IP router can be used to manage this complexity by controlling which hosts and which services may be accessed from remote locations.

In a typical application, host address filters allow remote stations to log in to a host that is known to have carefully administered usernames and passwords but prevent access to hosts that are less secure. Protocol filters allow logins, ftp and mail, but deny remote access to X11 or NFS.

All IP routers do packet filtering to reduce network load. Broadcast packets, packets for which the router does not have a route, packets with bad IP headers, and packets that have been bouncing around over too many gateways (packets with TTL = 0) are not forwarded [2].

Routers from several manufacturers can use IP source and destination address to provide administrative control of which hosts or networks may communicate with each other. Some also use UDP or TCP port or ICMP message type to control what applications network connections are used for [4, 14, 15].

Techniques for Secure Internet Gateways

Packet filtering is one of two common techniques for implementing a secure internet gateway. The other is an application-layer gateway that provides proxy access to the Internet. A third technique uses a combination of a packet filtering gateway and an authentication server.

Packet Filtering Gateway

Digital Equipment Corporation's *screend* is packet filter that runs as a UNIX process [13]. Packets are filtered on input. The decision to accept or deny a packet may be based on host address, subnet or application (port or ICMP type code). *Screend* supports one-way or bi-directional filters, source or destination addresses, and wildcards (accept **any** value). Figure 1 shows examples of filter specifications.

If the keyword **notify** is appended to a **reject** specification, *screend* will send an ICMP destination unreachable message to the source of a rejected packet. If the keyword **log** is included, *screend* will log each application of that packet specification.

Most of the code in *screend* is devoted to parsing the configuration file and building internal data structures. The syntax for the configuration language is specified with a BNF and implemented using *lex* and *yacc*.

Screend uses a cache of recently used packet descriptions and decisions to reduce lookup time.

Application-Layer Proxy

In the application-layer gateway used at AT&T Bell Laboratories described by Cheswick [6], two specialized machines are used: *inet* and *r70*. Only *inet* is visible to the outside world. It contains a very limited amount of secret information. For

```
from host xx.lcs.mit.edu tcp port 3
    to host score.stanford.edu tcp port telnet reject;
between host sri-nic.arpa and any accept;
```

Figure 1: Examples of Filter specifications

inbound connections, a challenge response authentication service is provided by *r70*. Most user accounts use this service rather than passwords. *Inet* is used to provide anonymous ftp and a store and forward mail router. In the outbound direction customized applications such a *ptelnet* and *pftp* send connect to inet via datakit or via Ethernet through *r70*. To hosts external to Bell Labs, these connections appear to come from *inet*.

The advantage of the application layer gateway is the addresses of hosts on the internal network are completely hidden from the external network. The disadvantages are that it is more complicated and it runs slower. The *inet* gateway is a MIPS M/120. File transfer rates peak at 44Kb per second. This is more than enough a for 56Kbps link, but not fast enough to take advantage of a T1 (1.54 Mbps) link.

Packet Filtering Gateway and Authentication Server

Several sites use a combination of a packet filtering gateway and an authentication server to secure an internet connection. The MITRE Corporation uses a *cisco* router to limit the number of hosts exposed to the Internet [8]. Applications such as telnet, rlogin, and ftp have been modified to use a *SecureID* smartcard system. Connections from the Internet are validated with a challenge/response system.

Strategies for Packet Filtering Gateways

There are two benefits from filtering packets: reduction in unneeded packet traffic and protection from unwanted, perhaps malicious use of network resources. How effective routers are at providing these benefits is largely a function of the flexibility and usability of the tools provided to the system administrator.

Routing Table Solutions

All IP routers decide to route or not route a packet based on the result of routing table lookup. In principle, the routing table could be used to decide to which destinations packets may be routed and to which they may not. This solution is secure if only static routes are used. Commonly used routing protocols such as RIP are not secure [9].

Some routers can choose from which source address they will accept RIP information [4, 11]. This helps secure against acquisition of incorrect routing information that was accidentally provided. However, because RIP information is passed in an easily forged datagram an unauthorized user to fool a gateway listening to RIP into adding a route.

A solution to this problem is to maintain a destination filter table of permit and deny rules in the same format as the routing table. In addition to routing lookup, the router looks in the filter table for the output interface and sends the packet only if a permit rule is found. In the case where the external gateway has only two interfaces (e.g., one 56Kbps and one Ethernet), this method can be used to limit traffic from outside the organization to a particular set of hosts. Because the filter table is separate from the routing table, it is secure against routing protocol packets and can only be updated by a user with system administrator privileges.

Version 1.0 of Telebit Corporation's *NetBlazer* provides such a destination filter facility. Because we were able to reuse the routing table maintenance and lookup code, it took us less than a week to implement. One benefit to system administrators that this approach provides is the routing and filter commands have a consistent syntax.

Input and Output Filtering

Filtering only on the output interface is often less than optimal. Consider a router that has a 56Kbps interface to the external network and several fast local area network (LAN) interfaces to internal networks. To control the flow of traffic without knowledge of the input interface requires filters be applied to LAN interfaces. Time spent in filter table lookup tends to reduce LAN-to-LAN packet throughput. If instead filtering is done only on the 56Kbps interface both in the input direction and the output direction, the same security objectives can be achieved without slowing down LAN-to-LAN routing.

Source Address Filtering

Some organizations apply one authentication scheme to connections within the internal network and another to connections from outside the network. Connections are considered internal if the source-destination pair is within the organization's internal address space. The integrity of this assumption is improved by applying a filter to the external interface that rejects packets in which the source and destination address are both in the internal network. This prevents an external host from avoiding more rigorous authentication by masquerading as an internal host.

Protocol Port Filtering

By looking at the destination port, the router can control which daemons can be accessed. Each of the TCP services, *smtp*, *nntp*, *ftp-data*, *ftp*, *finger*, *telnet*, *login*, and *shell* begins by connecting to a well-known socket which is listening to a port reserved for that service. The same is true for UDP destined for Sun RPC, RIP, and Domain daemons. By restricting the set of destination ports that may be accessed from the external network, system administrators may control which services may be accessed from the external network. One could, for

example, deny external UDP access to Sun RPC (e.g., NFS) and *routed* but allow domain name service by limiting UDP access to port 53. One could allow external access for mail and netnews by limiting TCP access to ports 25 and 119.

Special Cases

IP allows some special cases which make security through packet filtering a challenge: These include source routed packets and fragments.

Source routed packets may slip through a filtering router by appearing to be destined for an authorized host and then being forwarded to an unauthorized host. *Screend* solves this problem by not routing source routed packets [13].

Except for the first one, fragments of a packet do not contain the next level protocol information needed to do port filtering. Passing subsequent fragments is relatively harmless since it is difficult to compromise a system without sending it complete packets. On the other hand, one could flood a network with fragments. One option is to simply pass or reject fragments based on an address filter rule. This is what the *NetBlazer* does. Another is to simulate the reassembly algorithm by keeping a cache of previously seen fragments and match the decision for subsequent fragments with the one made for the first. This is what *screend* does.

The NetBlazer's IP Packet Filtering Facility

In addition to working correctly and efficiently we wanted the the *NetBlazer's* filter facility to be easy to use. We chose an internal data structure which made the per packet lookup processing simple and then put a lot of effort into providing commands to maintain these tables. To reduce code size and insure consistency, all filter commands use the same parsing function. Yet more than 50 per cent of the lines of filter source code are devoted to parsing command lines, almost 40 per cent are devoted to updating the tables, and only 10 per cent are devoted to the per packet lookup.

One-line Commands

Other implementations use configuration languages to create a filter configuration file. In these implementations, rules are applied to packets in the order in which they were entered [4, 13]. We wanted administrators to be able to update the filter table with one-line commands in much the same way that they update the routing table and we wanted the search process to be order of entry independent.

The *NetBlazer* IP filter facility contains the following one-line commands:

permit	adds a permit filter to an interface
deny	adds a deny filter to an interface
delete	deletes a filter from and interface
flush	remove all filters from an interface
lookup	tests a packet specification against the filter table
list	displays the filter table for an interface

What You See Is How It Works

When the *NetBlazer* administrator lists the filter table it displays the rules in the order in which they are applied. The *NetBlazer* also provides a lookup command which takes as arguments a source and destination address, an input interface, and optionally a protocol and destination port number. The command displays the results of input filter, routing, and output filter table lookup.

Lookup Order

Traditionally, a 32-bit IP address has been considered to have three components: network, subnet, and host [1]. Phil Karn's *ka9q* views an IP address simply as network and host with the network portion being of a variable length that is defined by the subnet mask [7]. Routing lookup is done by searching the network numbers with the longest subnet masks first. In this scheme host routes are treated as network numbers with 32-bit subnet masks. Routes are specified with a convenient *network/bits* syntax. For example

```
route add 143.191.10/24 en0
```

routes subnet 143.191.10 via the interface en0. When a packet comes in the first 24 bits of the destination address are compared to 143.191.10. If they match, the packet is sent out via en0.

The same lookup policy and specification syntax is used in the *NetBlazer's* filter facility. Except for the default behavior each filter rule has a network specification associated with it. Filter rules can specify a source network, a destination network, or a source network and a destination network (a source-destination pair). If the */bits* field is omitted, it's value is assumed to be 32 and the rule is applied only if the address is an exact match.

A Millisecond in the Life of an IP Packet

When an IP packet enters the the *NetBlazer*, the first test is to see if it was a hardware broadcast. Information in broadcast packets may be consumed by the *NetBlazer*, but it does not forward them. The second step is to determine if the packet is a valid IP packet. The packet is then tested against the input interfaces input filters (if any). If permission to route the packet is granted, routing table lookup is done. Having found a route to the destination address, the *NetBlazer* now knows the output interface. Output filter table lookup determines whether to send or reject the packet. Thus the *NetBlazer* forwards a packet only when the following conditions are true: (1) not a broadcast, (2) valid IP packet, (3) permitted by input filters, (4) a route to the

destination address exists, (5) permitted by output filters.

NetBlazer Examples

The following examples apply to a *NetBlazer* with a synchronous interface (*syn0*) to the *Alternet* and several local area network connections.

The filter commands shown in Figure 2 would limit inbound access to internal network to ftp, mail, news, and domain name service requests directed to the host ftp.telebit.com.

The *NetBlazer* permits shortening commands to the shortest unique abbreviation and specifying an IP address instead of a host name. The `permit` command in figure 2 could have been specified as:

```
p 143.191.3.1 syn0 tcp 20 21 25 53 119 i d
```

The list command displays the filter table shown in Figure 3.

The filter facility has an implied on/off switch which operates on a per interface per direction basis. Because no filters exist in the output direction, outbound traffic is not filtered. When the first filter is created, a complementary default behavior is created at the same time. Adding an outbound filter enables filtering in the outbound direction. For example, to forbid the transmission of any UDP packets from the 143.191 network:

```
deny 143.191/16 syn0 udp output source
```

The list command now displays the filter table shown in Figure 4.

This has two undesirable side effects: (1) domain name service from ftp.telebit.com is no longer available to the Alternet and (2) the default of *deny* is now applied to all outbound traffic. In a prototype that Telebit showed to some customers, there were separate defaults for input and output. Having more than one default behavior was sometimes useful, but often very confusing. The side effects can be corrected with the following two commands:

```
deny 143.191/16 syn0 udp !=53 output source
permit any syn0 output source
```

The list command now shows the results displayed in Figure 5.

The following filter command prevents an external host from spoofing the authentication server by pretending to be a host on the 143.191.1 network and sending a host route in a RIP packet to the gateway.

```
deny    143.191.1/24    143.191.1/24    syn0
```

The list command shows the results displayed in Figure 6.

```
filter
permit ftp.telebit.com syn0 tcp 20 21 25 53 119 input dest
```

Figure 2: Filter commands limiting inbound access

```
Source          Destination     Interface  Protocol  I/O   Permit/Deny
                143.191.3.1/32  syn0       TCP       In    Permit
                                port rules: =20 =21 =25 =53 =119
Default                         syn0                 In    Deny
```

Figure 3: Result of list command

```
Source          Destination     Interface  Protocol  I/O   Permit/Deny
                143.191.3.1/32  syn0       TCP       In    Permit
                                port rules: =20 =21 =25 =53 =119
143.191.0.0/16                  syn0       UDP       Out   Deny
Default                         syn0                       Deny
```

Figure 4: Filter table after forbidding UDP packets

```
Source          Destination     Interface  Protocol  I/O   Permit/Deny
                143.191.3.1/32  syn0       TCP       In    Permit
                                port rules: =20 =21 =25 =53 =119
143.191.0.0/16                  syn0       UDP       Out   Deny
                                port rules: !=53
any                             syn0                 Out   Permit
Default                         syn0                       Deny
```

Figure 5: Listing after correcting side effects

Because input or output was not specified, the *NetBlazer* created both an input filter and an output filter. The list command displays filter table entries in the order in which they are searched. Input filter lookup is done first then routing table lookup, then output table lookup. The most significant network number (the ones with the most bits) are searched first.

Performance

Studies indicate that traffic through routers tends to flow between pairs of addresses [10, 16]. While a cache size of two entries, Heimlich observes a hit rate of 0.48 doing wide area routing and 0.38 doing LAN-to-LAN routing. With a cache size of 16 entries, hit rates exceed 90 per cent [10].

The *NetBlazer* has a simple two-entry cache in which routes to the source address and destination address of the last packet are stored. If address-only filters are used, the appropriate filter tables are first checked. If permission to route the packet is granted the address is cached. The cache is not used when port filtering is enabled on either the input or output interface. Our observations of *NetBlazers* used to do Ethernet-to-Ethernet routing internally at Telebit find cache hit rates typically between 80 and 90 per cent. So far, we have not seen hit rates below 25 per cent.

Throughput was measured between two Ethernets with a filter table size containing four entries. The *NetBlazer* CPU is a 16-Megahertz Intel 386/SX. One way traffic from one host to another via the *NetBlazer* was varied until the maximum number of packets routed by the *NetBlazer* was observed. While this was being done, 100 ICMP Echo Requests and 100 ICMP Echo Replies per second were sent between a second pair of hosts to generate background traffic.

Maximum Total Packets Per Second Throughput		
Type of Filters	PPS	Hit Rate (Per Cent)
Address and Protocol	320	-
Address Only	440	77
None	470	81
Background Traffic 100 64-byte pings per second (200 PPS total)		
Foreground Traffic: 1500-byte packets		

Future Work

The *NetBlazer* needs to provide more flexibility in the way it deals with ICMP. It should distinguish between different ICMP packet types and provide customer selectable notification options including no notification and a choice of Destination Unreachable type including the new RFC1122 defined types shown in Figure 7.

We would like to spoof TCP connections with the *NetBlazer* and map one connection in to two. This would hide the the Internal Network from the outside world without requiring modification of application software. Options to filter on protocol source port and to log filter decisions are needed.

Conclusions

By providing powerful, flexible filters, the *NetBlazer* minimizes the number of interfaces the system administrator must deal with. One-line commands make it easy to modify, list, and test the filter set.

The *NetBlazer* uses simple internal data structures to provide security filters while maintaining a performance level that is at least 50 per cent as fast as routing without filters. A global two-entry cache can provide average hit rates that range from 25 to 90 per cent. By making this cache two entries per interface, performance can be further improved.

```
Source          Destination     Interface  Protocol  I/O   Permit/Deny
                143.191.3.1/32  syn0       TCP       In      Permit
                                port rules: =20 =21 =25 =53 =119
143.191.1.0/24  143.191.1.0/24  syn0                 In      Deny
143.191.1.0/24  143.191.1.0/24  syn0                 Out     Deny
143.191.0.0/16                  syn0       UDP       Out     Deny
                                port rules: !=53
any                             syn0                 Out     Permit
Default                         syn0                         Deny
```

Figure 6: Listing after correcting for spoofing

```
 9 = communication with destination network administratively prohibited
10 = communication with destination host administratively prohibited
```

Figure 7: Two new RFC1122 defined types

Author Information

Bruce Corbridge received an electronic engineering degree from DeVry Institute of Technologies in 1974. Over the past sixteen years, he has been employed as a test engineer and technical writer for several companies in Silicon Valley, including ISS Sperry Univac, Diablo Systems and Convergent Technologies. He is currently working as a technical writer in the Network Products division at Telebit Corporation in Sunnyvale, CA. Reach him via U.S. Mail at Telebit Corporation; 1315 Chesapeake Terrace; Sunnyvale, CA 94086-1100. Reach him electronically at uunet!telebit!bac or bac@telebit.com.

Robert Henig received an BSCS from Northeastern University in 1984. He then spent six years working for Intel Corporation in network programming and network management roles. He joined Telebit Corporation in Sunnyvale, CA in January, 1991 to develop software for the NetBlazer. Reach him via U.S. Mail at Telebit Corporation; 1315 Chesapeake Terrace; Sunnyvale, CA 94086-1100. Reach him via electronic mail at uunet!telebit!rhenig or rhenig@telebit.com.

Charles Slater received an MS in Social Science from the California Institute of Technology in 1980. After spending 10 years in support organizations helping various Silicon Valley companies work around flaws in network products, he decided to go to work for a manufacturer to see if he could do a better job than his former vendors. He has spent a little more than a year Telebit Corporation in Sunnyvale, CA writing software for the NetBlazer. Reach him via U.S. Mail at Telebit Corporation; 1315 Chesapeake Terrace; ; Sunnyvale, CA 94086-1100. Reach him via electronic mail at either uunet!telebit!cslater or cslater@telebit.com.

References

[1] Braden, R.T., "Requirements for Internet Hosts - Communication Layers RFC 1122", October 1989

[2] Braden, R.T.; Postel, J.B., "Requirements for Internet gateways. RFC 1009", June 1987

[3] Bradner, Scott O., "Testing Multiprotocol Routers: How Fast Is Fast Enough?", Data Communications, February 1991, pp 70-86.

[4] cisco Systems, Inc., "Gateway System Manual/Software Release 8.2", Menlo Park, CA, November 1990.

[5] Carlin, Jerry M., "Internet Gateway Security Checklist", USENIX Security II Workshop, Summer 1990, pp. 145-147.

[6] Cheswick, Bill, "The Design of a Secure Internet Gateway", USENIX Anaheim Conference Proceedings, Summer 1990, pp 233-237.

[7] Ford, G. E., "Beginner's Guide to TCP/IP on the Amateur Packet Radio Network Using the KA9Q Internet Software", Version 1.0, May 9, 1990

[8] Goldberg, David S., "The MITRE User Authentication System", USENIX Security II Workshop, Summer 1990, pp. 1-4.

[9] Hedrick, C.L. "Routing Information Protocol", RFC 1058, June 1988

[10] Heimlich, Steven A., "Traffic Characterization of the NSFNET National Backbone", USENIX Washington, D.C. Conference Proceedings, Winter 1990, 0.25i7-227.

[11] Honig, Jeffrey C., "Gated(8)" manual, Cornell Theory Center, Cornell University, Ithaca, NY January 1989.

[12] McNeill, Keith, "SUMMARY: Need firewall telnet/ftp gateway", Electronic Newsgroup: alt.security, May 8, 1991.

[13] Mogul, Jeffrey C., "Simple and Flexible Datagram Access Controls", USENIX Baltimore Conference Proceedings, Summer 1989, pp 0.25i3-221.

[14] Mogul, Jeffrey C., "Re: well-behaved firewalls", Electronic Newsgroup: comp.protocols.tcp-ip, June 25, 1991.

[15] Nussbacher, Henry, "Comparison of Multiprotocol Routers", Version 1.7, Electronic Mailing List: tcp-ip@nic.ddn.mil, November 1990

[16] Paxson, Vern, "Measurements and Models of Wide Area TCP Conversations", Computer Systems Engineering Department, Lawrence Berkeley Laboratory, University of California, Berkeley, CA, LBL-30840, May 1991.

[17] Postel, J.B., "Internet Protocol. RFC 793", September 1981.

[18] Postel, J.B., "Internet Message Control Protocol. RFC 792", September 1981.

[19] Postel, J.B., "Transmission Control Protocol RFC 793", September 1981.

[20] Postel, J.B., "User Datagram Protocol. RFC 768", August 1980.

Cloning Customized Hosts (or Customizing Cloned Hosts)

George M. Jones & Steven M. Romig - The Ohio State University

ABSTRACT

In an environment consisting largely of many similar machines, it is certainly advantageous to treat these machines as clones of a master copy (the so-called "cookie cutter" approach). One of the problems with this approach is that vendor supplied tools typically do not allow the system administrator to easily change the master copy or to customize individual hosts after they have been cloned. In this paper we will discuss a general methodology for handling clones that includes provisions for variations among the machines. We will also discuss `update-client` and `build-server`, two tools that exemplify this methodology in the OSU-CIS environment. Finally we will analyze the advantages and disadvantages of these tools and describe our future plans.

Introduction

We have a fairly large number of nearly identical hosts at The Ohio State University Computer and Information Sciences Department (OSU-CIS). We have 260 diskless Sun SLCs (our Sun clients), several SparcStation 1s (used as clients), 22 Sun file servers (20 3/180s, 1 4/280, 1 4/330), 25 SparcStation 2 file servers that are slowly replacing the Sun 3/180s and the 4/280, and about 2 dozen other UNIX hosts. The Sun clients are nearly identical in their hardware configuration. Most of the clients are SLCs with 8 megabytes of memory, and most are diskless. Most of the 3/180 servers have identical SMD disk configurations, the rest have nearly identical SCSI disk setups. All of the SparcStation 2 servers are identical. We have been using the "cookie cutter" approach since we installed our first Suns (three Sun 2s) long, long ago.

Initially we used Sun's standard tools, which are now called `suninstall` and `add_client`, to setup servers and their diskless clients. As our environment grew more complicated we discovered that the tools that Sun provided were insufficient for maintaining our environment for several reasons.

Both `suninstall` and `add_client` were designed to install a SunOS distribution on a wide variety of platforms at all Sun sites. The tools are very general and require a certain amount of expert knowledge to get the configuration of a given server or client correct. This is fine for a tool that is used once in a while by a trained administrator to install a distribution for the first time. However, it is difficult for a relatively untrained operator to use these tools to correctly setup a server or client (as they often have to do to fix things). Ideally one would like `suninstall` and `add_client` to automatically use predefined configuration files to reduce the number of critical choices that untrained workers have to make in setting up servers and clients.

It is also difficult to change the "master copy" (or prototype) that `suninstall` and `add_client` use to create servers and clients. In the case of `suninstall` the prototype is the SunOS set of distribution tapes (or worse, the unwriteable CD!), and it would be unwieldy at best to make changes to `/bin` or whatnot and rewrite the tapes. `add_client` uses a prototype root directory (stored on the server) to create new clients. For most of the changes that we make to the clients, it would be quite easy to simply change the on-line prototype and be done with it. However, some of the changes would require changing the `add_client` script itself - most notably, we use a file called `/etc/rc.config` to contain host and network information (addresses, default gateways, broadcast address, netmask). It would be nice to be able to create a new prototype that included local changes to the standard distribution.

Further, `suninstall` and `add_client` make no provision for individual variation. Although we have 260 SLC clients that are essentially alike in most respects, 66 of them have significant changes that would have to be reinstalled by hand after running `add_client`. Some have printers and require special printcap files, some have swapping disks and require special kernels, some have special daemons that run on them, some have different `/etc/ttytab` files and so on. Similarly, our servers are alike in most respects, and yet different in some subtle but important ways (special daemons, printers and so on). We could handle this in part by creating shell scripts to make those specific changes to the standard client or server layout, and then run the shell scripts as post-processing after using the Sun tools.

In summary, `suninstall` and `add_client` are insufficient for the task of building many cloned servers and clients because they do not allow you to make your own prototype, they do

not provide for automatic customization of the hosts being installed, and because one cannot easily select a predefined configuration and apply it to a host.

A General Approach To The Problem

We would like to clarify that we are not trying to solve the problem of setting up a server or client for the first time. `suninstall` and `add_client` do this work admirably - they are fairly easy to use, and they are flexible enough to handle almost any sort of configuration.

The problem we are trying to solve is this: having installed the software and made our local changes, we want to be able to create a prototype out of the system we have set up, and use that prototype to setup subsequent servers (or clients), or to repair damaged servers and clients.

There are two parts to our proposed solution. We need a tool to create a prototype based on an existing server or client which is to serve as a model for subsequent clients and servers. We also need a tool to build a server or client based on that prototype. The building tool should allow one to easily select the configuration to use, and it should have a provision to use host-specific customizations that will allow a site administrator to record and retain individual variation in a collection of hosts based on a common prototype.

At OSU-CIS, we have created tools that conform to this model to solve this problem for diskful servers and diskless clients. The rest of this paper describes the tools for building servers and clients (respectively) and then summarizes our experiences with these tools and our future plans for them.

The Build-Server Script

Build-Server Overview

The `build-server` script is used to load a standard set of software onto Sun fileservers from a standard software prototype. We form the prototype by using `suninstall` to install the SunOS distribution on a server. We add our local modifications and test the system until it passes muster. Then we save a copy of the `/` and `/export` file systems with GNU `tar`. These `tar` files constitute the prototype from which other servers are created (or fixed).

To build a new server, the machine that is to have software loaded onto its disks is booted as a diskless client (though it would be possible, in principal, to boot it off a portable disk and use that to do the update). Once the server is running as a client, the `build-server` script is run to load software onto the disks.

The script knows how to do disk formatting, surface analysis, disk partitioning, create filesystems, check filesystem consistency, load software onto the partitions, select and install default kernels and how to install boot blocks. Based on user input, it can do as little as load software back onto one partition (for example, if a partition was badly damaged and had to be newfs'd), or as much as analyzing, formatting, partitioning, and loading software onto all disks on the system (for example, when we setup a new server).

Making a Master for Build-Server

Making a prototype for the `build-server` script consists entirely of saving copies of the "important" file systems (such as `/` and `/export`) with GNU `tar`. These copies should be made from a server that is up to date and which has been thoroughly tested, since any bugs present on the server will be saved in the copies and inflicted on servers created from them.

Build-Server Configuration

The `build-server` script uses a configuration file that allows the user to establish a mapping between a server's hostname and a particular server configuration. Among the elements of a configuration that may be specified are the architecture type, the number and type of network interfaces, the number and type of disks, the partitioning to use, and the path to a nonstandard kernel to use.

All servers are created equal, but they do not stay that way very long. Although the servers are built from the same prototype files, there are various changes that the `build-server` script makes to create their identity and to install special differences (like services that run on one server but not another).

The first changes that must be made to all servers are the host name and network address. We have created a configuration file, `/etc/rc.config`, that defines shell variables for items that vary from server to server, such as the hostname, network address, names of network interface(s), etc. This is used by all of the other RC files, and allows us to gather most of the host specific information into a single file where it can be easily edited. One of the first things that the script does after loading the root filesystem is to modify the values of these shell variables (`ed(1)` is your friend!) to reflect the proper values for the particular server in question.

We maintain identical `/etc/rc` and `/etc/rc.local` files on all servers. Boot time actions that are particular to certain servers are invoked from `/etc/rc.local`, keyed off the host name.

Benefits, History and Experience with Build-Server

The `build-server` script collects expert knowledge of system configurations and the processes necessary to perform system setup in a form that both documents them and makes it possible for nonexperts to perform the processes. Some of the tasks that are automated are tedious and error prone if performed manually. Other tools such as `suninstall` have similar benefits but do not make it easy for a non-expert to choose and apply various configurations to particular hosts.

The original version of `build-servers` was written when we were faced with the task of converting 20+ servers from supporting 250 Sun 3/50s running SunOS 3.5 to 250 SparcStation SLCs running SunOS 4.0. All of the servers were converted in a two week period by two people, with as many as four updates running in parallel (one has to be mindful of resource contention limited items such as tape drives, and of network limitations if one is running more than a single copy of `build-server` simultaneously!). The script is currently being updated to load software onto the SparcStation 2s with SCSI disks. The script itself required almost no modification. It was sufficient to simply write new configuration file entries.

Problems With Build-Server and Future Plans

`build-servers` attempts to be a user-friendly, easy to use interface for doing most initial system configuration tasks (after the hardware is set up), but reformatting a disk is still a big deal, running 10 passes of disk analysis still takes a long time, and it will always be possible to screw things up in a big way if such tools are not used with care by people who have a certain level of competence and training.

`build-servers` does not replace `suninstall` - it is not nearly so versatile. Its sole purpose is to apply stored configuration information and a changeable software prototype to repair or install diskful servers with optional host specific customizations. It also does not handle the needs of subsequent updates once the servers are running - that is what our update-servers script is for (see below).

Currently, the process of setting up a server to boot as a client in our environment is not trivial. It requires a good deal of knowledge of things such as tftpd, rarpd, Ethernet addresses etc. It is not currently feasible to tell one of our operators "go boot server X off of server Y." Also, adding new servers to the network (not just rebuilding old ones) is somewhat involved. There are a large number of changes that must be made to the nameserver databases, `/.rhosts`, `/etc/hosts.equiv`, `/etc/fstab`, NIS (nee YP) databases, etc. Something almost always gets missed on the first pass. This process could stand some automation. We are planning to set up the SparcStation 2s so that they can be easily be booted as diskless clients from other servers, from which you could run the `build-server` script to fix any problems.

We intend to add support for scripts specific to a particular server to be run after the standard software is loaded onto the server. This will allow for further customization of individual servers while retaining the benefits of having a common starting point and being able to reload a server at any time.

The script currently only loads system software and locally installed software packages, not user files. We intend to add the ability to have it load user files from our regular backup tapes.

We may move away from using tape for the master copy and keep a copy on disk updated at regular intervals.

In the longer term we may consider adding some "intelligence" to the script so that it can automatically check the sanity of things like the contents of the root filesystem and reload the software if needed. If a file system goes up in smoke, it still takes a certain amount of expertise to determine whether one should reload the file system from scratch to fix it, or just restore a few missing files from the latest backups.

The Update-Client Script

Update-Client Overview

The `update-client` script is used for setting up diskless sun workstations. It uses a client root prototype (stored as a GNU `tar` archive of what should be in a client's root directory), a client configuration database (the `/etc/client-info` file) and optional scripts that are used to customize specific clients.

To build a new client, one need only setup the prototype (with the `make-master` script), edit the client configuration file, and run `update-client clientname`. The `update-client` script will remove the client root directory (if it exists), create a fresh root from the master, setup the various architecture and host specific features according to the `/etc/client-info` file, and execute the client customization script (if it exists). The process is simple and foolproof and results in a client root that is set up the way that it "is supposed to be". Our operations staff frequently uses this command to recreate broken client partitions.

Making a Master For Update-Client

A script called `make-master` is used to create a prototype GNU `tar` copy of the root directory of a client that is to serve as the prototype for the others. The `make-master` script copies a named client's root directory, sanitizes it by

removing log files and architecture specific things (vmunix, sbin and so on), and uses GNU tar to save a copy. The script will refuse to make a master if the client that is named has a customization script (see below), since one would not want to propagate those customizations through the master copy to the rest of the clients. The saved prototype file is stored in a common location on all servers, where update-client can find it. The same prototype file can be used for Sun clients of any architecture, since the architecture specific changes are simple and are easily handled by the update-client script itself.

Update-Client Configuration and Customization

update-client gets its host information from the /etc/client-info file. The client-info file describes common characteristics of each of the clients, such as hostname, the location of its root directory and swap file on its server, which server it boots from, and its binary and kernel architectures. Here is an excerpt from the client-info file on a typical server:

```
KEYWORDS server root swap arch karch\
  fstab proto swapsize time broadcast

DEFAULT
  arch      sun4
  karch     sun4c
  fstab     /usr/local/config/fstab
  proto     /usr/local/config/master.gtar
  swapsize  30m

#
# Fish entries
#
DEFAULT
  server    fish
  time      0
  broadcast 128.146.29.255

carp.cis.ohio-state.edu
  root      /export/clients0/carp
  swap      /export/clients0/carp.swap
bass.cis.ohio-state.edu
  root      /export/clients0/bass
  swap      /export/clients0/bass.swap
```

This example shows most of the features of the client-info file. The keywords entry names the fields that are allowed to appear in the file. This is used for some simple error checking. The default entry sets default values for fields. In this case, we set the default binary and kernel architectures, the location of the fstab file and the prototype root, and the size of the swapping file. In a second default entry, we set the server, time (used when making crontab entries that run on all clients), and the broadcast address. The default entries can be used repeatedly to set new defaults throughout the file. Finally we have the client entries. Each entry starts at the beginning of a line in the file, and contains of all the name/value pairs that appear until the next entry starts. This file essentially lists all of the information needed to create a client root and swap for it to boot.

In addition to the normal client information recorded in the client-info file, a client can also have special customization information stored in a shell script in the /usr/local/config/specific directory. When update-client finishes the root directory for a client, it looks for a shell script in the specific directory with the same name as the client. If it exists, it is executed. The shell script is run with its current working directory set to the client's root directory, and can do anything that it wants to that client's files. For example, Steve has a client customization script for his workstation that rewrites the /etc/fstab file to remove most of the NFS mounts, rewrites /etc/rc to start amd (an automounter), and installs a kernel that supports the SCSI disks, tapes and CDROM that are attached to his SLC. The client config scripts are saved between client updates, so if you want to make permanent changes to a client, you simply create a customization script that does the right things, and subsequent client updates will retain your changes on top of the new client setup.

Benefits, History and Experience with Update-Client

Update-client is intended to be easy to use, and since the current version only takes one argument (the name of the client to update) it is hard to get it wrong. The operations staff at OSU-CIS uses it frequently to fix broken clients. It also provides for a fairly simple means for various people on the staff to set up customized workstations - we have customization scripts for 66 clients at this time. It also allows us to easily and thoroughly make changes to the root directories of all of the clients - it takes about 45 minutes to update the 260+ client root directories.

Problems with Update-Client and Future Plans

Although update-client does work well, it still has its problems. The main problem is that there are still some bits of configuration information in the script itself. For example, there is a fragment of the base fstab file used on clients, and parts of the base crontab file. We have to edit the script itself to change these and a few other things. It is currently being rewritten to move yet more of the configuration information out of the script itself.

Update-client is not a replacement for Sun's add_client script. It is not versatile enough to use for the task of creating new client setups, especially on new releases of the OS.

Miscellania

When a newly created server is booted for the first time a script called `update-all-clients` is run automatically from `/etc/rc` to set up any diskless clients that that server has. `update-all-clients` simply gets a list of the names of this server's clients and runs `update-client` on each to create its root directory and swapping file.

Once we have created a bunch of nearly identical servers, we also need a mechanism to keep the software consistent. We use a shell script called `update-servers` to propagate changes from a "master" server (typically the one used to create the software prototype used by `build-server`) to the other servers. The `update-servers` script is a "smart" wrapper for `rdist(1)`. It accepts directory names on its command line and checks to see that it is being run from the correct host, checks for unsafe directories or files (such as `/etc` - it would be bad if the master's `rc.config` file were pushed to the other servers!), creates a `rdist` script to update those directories, and invokes `rdist` to update the servers.

Concluding Remarks

These scripts have proven to be incredibly useful. They help us maintain an environment where the various hosts are constructed from a consistent base (the clone concept) while allowing us the freedom to introduce individual variations. They make it possible for relatively untrained staff members to build new servers and clients. The ability to make new prototypes is also very useful, but the provision for preserving and applying customizations has probably been the biggest advantage.

We are continuing to improve both of these tools by making them more general and by moving most of the configuration information out of the scripts and into configuration files. We do not know whether these specific scripts would be suitable as they are for use at other sites or not. It should be easy to modify them to work in other environments.

In the long run, it might be better to add configuration selection and customization to `suninstall` and `add_client`, which would eliminate the need for extra tools. One could then use `add_client` in the usual way to create an initial client, change it to reflect local requirements, and make a master prototype from that client. Then one could use `add_client` with different options to apply that prototype and any desired customizations to other clients to build or fix them. Similar things could be done with `suninstall` so that one could either answer a bunch of questions and make a new server configuration, or apply an existing prototype and customizations to a server to repair or build it.

Acknowledgments

Steve Romig wrote the original specifications for both `build-server` and `update-client`. Tom Fine wrote the original `build-server` script. George Jones worked on it with him, and Tom is working on the newest version. Steve wrote `update-client` and has been slowly and continually improving it. Various members of the OSU-CIS staff have made numerous suggestions, requests and helpful comments about both scripts.

Author Information

George Jones is a member of the software support staff at the CIS Department at The Ohio State University. His email address is george@cis.ohio-state.edu, his U.S. Mail address is The Ohio State University; Department of Computer and Information Science; 2036 Neil Avenue Mall; Columbus, OH 43210, and his telephone number is 614-292-7325.

Steve Romig is the software staff manager for the CIS Department at The Ohio State University. His main professional interests are in simplifying and automating system administration tasks and in computer security. His electronic mail address is romig@cis.ohio-state.edu, his U.S. Mail address is The Ohio State University; Department of Computer and Information Science; 2036 Neil Avenue Mall; Columbus, OH 43210, and his telephone number is 614-292-8018.

References

Sun Microsystems, SunOS Reference Manual Volume i, section 1: "rdist(1)", July 17, 1986.

Sun Microsystems, SunOS Reference Manual Volume iii, section 8: "add_client(8)", January 13, 1990.

Sun Microsystems, SunOS Reference Manual Volume iii, section 8: "suninstall(8)", January 13, 1990.

Sun Microsystems, SunOS 4.1.1 Release & Install, "Installing the SunOS", 1990

NAME

build-server - build a fileserver

SYNOPSIS

build-server

or

build-server [**options**] **name_of_server_to_build**

Options are:
-update_server name_of_update_server
-tape_drive rst[0123]
-working_dir where_to_find_format_files
-disk [0123]
-partition partition_name_or_device
-silent
-noformat
-noanalyze
-nopartition
-nonewfs
-norestore_files

DESCRIPTION

The *build-server* script is used to format and analyze server disks, load the standard set of server software onto the disks and perform the necessary customization to change the servers name, addresses, etc and to install an appropriate kernel and boot block.

The basic mode of operation is to recable the server and boot it as a diskless client of some other server, to log in to the server-cum-client as root and to run *build-server* with the appropriate options.

To perform the update you will need to load a master server tape, as created by *make-server-tape(8)* into the tape drive of some system (the *update_server)* that is accessible over the network. The server-cum-client must be able to rsh as root to the update_server in order to be able to load the files.

Once *build-server* is finished loading files off the tape the server should be recabled and booted multi-user. If everything worked correctly the root filesystem will have been built for the proper architecture, a usable kernel and bootblock will have been installed so it should "just boot".

The first time that the newly [re]built server boots multiuser it will automatically run a *update-all-clients(8)* to build the appropriate clients.

Also, once new server is up multiuser you should do an

update-servers -server new-server-name -force all

to install copies of software that are more up-to-date than the software on the tape from which the server was built.

Lastly, with the appropriate use of switches *build-server* can be used to perform any part of the server building process all the way from building a new server starting with unformatted disks (the default behavior) down to reloading software onto an individual partition (say, after a filesystem gets badly corrupted).

The *name_of_the_server_to_build* parameter must be in the server-cum-client's /etc/hosts file (this is where build-server gets the IP address).

USAGE

Build-server may be invoked one of two ways: with arguments or without. If it is invoked with no arguments it will prompt for all needed information (name of server to build, which disks to build, etc). If it is invoked with arguments (the name of the server at a minimum) it will build the server as requested using defaults found in the configuration file (*build-server.conf(5))* with the arguments provided serving to override defaults (where to find the update tape, etc).

OPTIONS

-update_server name_of_update_server

This switch may be used to specify the system which the tape drive lives. The default is "favorite".

-tape_drive tape_drive

This switch allow you to specify which drive to use. Legal values are rst0, rst1, rst2, and rst3. The default is rst0.

-working_dir path

This switch allows you to specify an alternate path in which *build-server(8)* will look for input files for *format(8).*

-disk disk_number

This switch is used to specify which disk(s) are to be rebuilt. By default 0 and 1 are built. Legal values are 0, 1, 2 and 3. Multiple disks may be specified with multiple uses of this switch. If any disks are explicitly specified with this switch the defaults are ignored (i.e. specifying "-disk 1 -disk 2" would result in disks 1 and 2 being reloaded, but not disk 0 or disk 4).

-partition partition_name_or_device

This switch is used to specify which partitions(s) are to be built. The default is every partition on the disk (as specified in the config file). If any partitions are explicitly specified via this switch then only those partitions are built.

By default the output from format(8) is displayed. This switch
suppresses that output.

-noformat

By default all disks are formatted. This switch suppresses formatting.

-noanalyze

By default all disks have surface analysis run. This switch suppresses analysis.

-nopartition

By default all disks are partitioned. This switch suppresses partitioning.

-nonewfs

By default new filesystems are created and fsck(8)ed. This switch suppresses the creation of

new filesystems (not usually a good idea unless you are really confident in the integrity of the existing filesystems, and, hey, what the heck, this is a fairly low cost operation time-wise, so why not do it anyhow ?)

-norestore_files
Don't load any files onto the filesystems. If you use this option you don't have to worry about loading a tape anywhere or being able to rsh to the update-server, but you don't get any files either.

EXAMPLES

The following example would build the server "fish" from scratch using the default build_server (favorite) and tape drive (rst0).

```
build-server fish
```

The next example would build the server "muppet" using an alternate tape drive and update server.

```
build-server -update_server seventh -tape_drive rst1 muppet
```

The final example would format, analyze, partition, newfs and fsck muppet's disk 1 without restoring any files.

```
build-server -disk 1 -norestore_files  muppet
```

FILES

/usr/local/utils/build-server/build-server
The script
/usr/local/utils/build-server/build-server.conf
The configuration file
/usr/local/utils/build-server/*.disk[0123].{format,partition,analyze}
Input files for *format(8)* to do various things to various disks.
/usr/local/utils/build-server/format.dat
format.dat for *format(8)* that understands all our local disk types and partition layouts.

SEE ALSO

make-server-tape(8),change_architecture(8),update-servers(8), update-all-clients(8),rdist(1)

BUGS

There is still too much hacking of files that needs to be done by hand at various places on our network to add a new server, specifically, adding it to the nameserver configuration, .rhosts, /etc/hosts, /etc/client-info, yp groups (netgroup, lists), and so forth.

NAME

update-client - remake a client's root partition

SYNOPSIS

update-client client-name

DESCRIPTION

update-client is a complex script which creates client root partitions on a server.

Clients are created from a master client prototype (/usr/local/config/common/master.gtar) using special information gathered from /usr/local/config/client.defaults and /etc/client-info, and from client specific config files (/usr/local/config/specific/ClientName, if it exists). In brief, **update-client** removes the existing client directory, uses the master prototype to create a new one, and edits it in various ways to change the architecture (according to info found in /etc/client-info) and host name, host address and serve. This procedure creates a "standard" client according to the prototype definition. If the client specific config file exists, it is executed to make any additional changes.

At OSU CIS, the files in /usr/local/config/common are used to construct or replace the standard client root partition. *master.gtar* is a sanitized copy of a standard client created with GNU tar (and the **make-master)** script. *fstab.client* contains the stock fstab entries that should be inserted in a client's fstab file.

The *update-client* script restores the root file system, copies the latest replacements in, and uses *ed(1)* to hack on several files to tailor things to suit the server and the client (changing the hostname, domain-name, default gateway, and so on).

The files in */usr/local/config/specific* are assumed to be scripts for tailoring specific clients. If a file with the same name as the client being updated exists, that script will be run after *update-client* finishes the rest of the configuration. This feature can be used to tailor particular clients to suit particular needs (for example, to enable logins on a serial port). The scripts should be written assuming that they will be run as root, with the current working directory set to the client's root directory. Be extremely careful about this - if you go hacking on /etc instead of ./etc, bad things will happen to the server (and you)!

See the script itself for more details (the ultimate documentation - at least I commented it...).

The */etc/client-info* file contains information about the clients. Currently, it is used to indicate where the client's root and swap are, and what architecture and kernel architecture the client should use. We probably ought to move more of the info from /usr/local/config/client.defaults to this file, but haven't yet.

The client-info file contains entries for each client. Each entry starts with a client name (with no indentation), followed by name/value pairs that specify various attributes of that client. For example, "foo arch sun3 karch sun4" would be a client named foo with strange architectures. You can set default name/value pairs throughout the file: this is frequently done for the "server" field.

FILES

/usr/local/config/client.defaults	defines default settings
/usr/local/config/common	configuration files common to all clients
/usr/local/config/specific	configuration files for particular clients
/etc/client-info	More client information

SEE ALSO

update-all-clients(8), make-master(8), client-info(5)

Staying Small in a Large Installation: Autonomy and Reliability (And a Cute Hack)

Edward Wang - University of California, Berkeley

ABSTRACT

This paper describes how a small group within a large organization has kept both the autonomy and reliability of a small installation and some of the benefits of being part of a large, professionally managed installation. In particular, I will describe the system we use to replicate locally the frequently used parts of a large software warehouse (a large, read-only, NFS-mounted file system). This allows us to use all of the available programs with very little effort, while remaining largely immune to remote server crashes and preserving the performance of a private file server.

1. Introduction

The purpose of this paper is twofold: to describe one of the tricks we use to manage our machines, and to extol the virtues of being small.

Section 2 describes our environment and how we manage the partially-replicated file system. Section 3 reports our experience with the system, its current status, and some of the remaining flaws and inefficiencies in the implementation.

While I am writing from the perspective of a user, some of the ideas in this paper may nevertheless be of interest to managers of large installations.

2. Tapping a Large Installation

We are a small research group with five Sparcstation IPCs in the Berkeley Computer Science Division. After acquiring the machines, we gradually weaned ourselves from all outside servers. We have enough disk space for everything we need. One of the machines provides mail and NIS services, and an NFS-mounted `/usr` partition. Within our small world, we have the reliability of few components and close proximity, and the freedom of autonomy, and we save money by doing things ourselves (over $600 a month)[1].

The CS Division offers a large repository of public-domain and site-licensed software (the Software Warehouse, or SWW). SWW has binaries for several machine architectures. All the programs are configured to run from the file system `/usr/sww`, which is to be NFS mounted read only on all machines that wish to use it.

SWW is convenient and free. Using it relieves us of installing large software packages. We get almost every available package, already configured and compiled, and they will be updated automatically (as far as we're concerned). We can also report problems to a professional staff. On the down side, the SWW server is not necessarily very reliable, and with many clients, it may be slow. The question is how to keep what we have and still use SWW.

The solution is to copy parts of the `/usr/sww` hierarchy. (We don't have enough space for all of it.) These parts (presumably files we need for daily work) will be on our own server, therefore as reliable as ever.

Because all programs are compiled to run from the path `/usr/sww`, the local copies must shadow the remote files in place. We therefore have to mount the remote file system under a different name (`/usr/swx`), make the copies into `/usr/sww`, and fill in the gaps with symbolic links (from `/usr/sww/...` to `/usr/swx/...`)[2].

A shell script (*rd*) together with *rdist* manage the copies and links. When properly configured, running *rd* updates the copies when the originals change, and adds links to new files and directories.

A configuration file specifies the files and directory trees to be replicated. These are the lines we use to copy Emacs:

```
bin/emacs
lib/emacs
share/lib/emacs
!share/lib/emacs/info
```

[1] For example, we do our own backups. An automated dumper runs nightly, following a variant of the schedule recommended by Sun. The only manual intervention is to insert the correct dump tape into the drive each day.

[2] The Translucent File System may have been able to perform this task. However, it is an untried system, while symbolic links are better understood and known to work reliably.

```
!share/lib/emacs/man
```

The binary `bin/emacs` and the directory trees `lib/emacs` and `share/lib/emacs` are copied, but excluding `share/lib/emacs/info` and `share/lib/emacs/man`, which will be linked.

The script *rd* reads the configuration file and performs the update in two phases. First, a `distfile` is constructed and passed to *rdist* to make the copies (if necessary). Next, the directories where links may need to be made are scanned and compared against their counterparts in `/usr/swx`. Links are made if necessary, and extra files and links are removed. The directories to scan are computed from the list of files and directories copied. In the Emacs example, if those are the only files to copy, *rd* will scan these places: `bin`, `lib`, `share`, and `share/lib`, as well as `/usr/sww` itself.

Command-line arguments can limit the scope of the update to a subset of `/usr/sww`. There is also a *verify* option to suppress actual modifications.

3. Successes and Failures

A very manageable configuration file (about 100 lines) specifies our current setup. We use less disk space than before, because only parts of some large packages are copied. In all, we copy only about 100 megabytes of the one to two gigabytes in the full SWW. An update with nothing to do takes about 10 minutes[3]. I test the configuration by secretly unmounting `/usr/swx` while a user is logged in and working. It almost always succeeds.

When the SWW server does go down, accessing an uncopied file still causes annoying delays and NFS timeouts. The current NFS implementation doesn't seem to allow such a file system to be unmounted.

Rdist is not the perfect tool for this job. When a file is deleted from the configuration file, it must also be removed manually before *rd* will run correctly. Updating the executable of a running process causes the process to fail. I have resorted to running verifying passes periodically from *cron* and updating passes by hand. It would be better to have a custom implementation of *rd* in C.

Acknowledgements

I thank the people in my group for putting up with my fiddlings in the early days of *rd*. I also thank Marcia Feitel, Kinson Ho, and Luigi Sememzato for correcting the first draft of this paper.

[3]Actually, this doesn't include some font directories that are snake pits of thousands of files. We bother to update these only very infrequently. A complete verify pass takes about 20 minutes.

Author Information

Reach Edward Wang at the Computer Science Division; University of California; Berkeley, California 94720. Write to him via electronic mail at edward@ucbarpa.Berkeley.EDU.

Some Useful Changes for Boot RC Files

Steven M. Romig - The Ohio State University

ABSTRACT

Virtually every system administrator has run into problems wherein a host will not boot correctly. The standard RC files (`/etc/rc` and `/etc/rc.local`) distributed with most systems seem to have been written to obfuscate the booting process, and hence to hinder the process of debugging the boot procedures. This paper describes several simple changes that one can make to the RC files on a system that should ease debugging and administration tasks. These changes should be applicable on any system that uses shell scripts to control the initialization and startup of system services at boot time.

Introduction

Fixing boot problems on a host can be a frustrating experience for a number of reasons. In the standard RC files supplied with most BSD derived systems the daemons are announced (through `echo`) after the daemons are started. If a daemon hangs for some reason, the corresponding echo message will not be displayed, and the last message on the console will only tell you what was last successfully started, not what is hung. Unless you have a copy of the RC file on hand, your only recourse is to halt the system, reboot it single user mode and read the RC files to determine what would have been run after the last message that appeared.

Since some RC files are run in an environment where stdout and stderr are directed to `/dev/null` (rather than `/dev/console`), any error messages or diagnostic output that might have been helpful is discarded. The only recourse in most of these situations is to do without, or hope that `syslogd` has been started already and caught some messages. One could change the RC files to redirect stdout and stderr to `/dev/console`, but this can only be a temporary solution since it is often best that system daemons start up with no controlling terminal.

In situations where you have many similar hosts it would be nice to be able to minimize the differences between the RC files of those hosts so that you can maintain and test them more readily.

Finally, most vendors have been using the `/etc/rc.local` file for their local changes to the system, rather than reserving it for changes that are really site specific (e.g., local to the user). This makes it difficult to separate site specific changes from vendor specific changes and complicates one's work at system upgrade time.

To address these problems, we have made some simple but effective changes to the RC files (`/etc/rc.boot`, `rc.single`, `rc` and `rc.local`) on our Sun systems (currently running SunOS 4.1) that have largely eliminated the grief we usually associated with debugging boot-time problems on the Suns. It should be easy to apply similar changes to the RC files from other vendors as well.

Switch the Order of Echo and Commands

The first change we made was to simply switch the order of the echo messages with respect to the commands that they describe. Most vendors ship RC files that place the echo commands after the service that they describe is started:

```
if [ -f /etc/somefile ];
then
  /etc/someservice
  (echo 'someservice')>/dev/console
fi
```

This makes debugging the boot process more difficult since if something hangs, you only get the message for what was last started, not for what hung. So we simply switched the order of the echo and the service:

```
if [ -f /etc/somefile ]; then
  (echo 'someservice')>/dev/console
  /etc/someservice
fi
```

This way we always know what is about to be started. If we wanted to be really fancy we could change that to something like this:

```
if [ -f /etc/somefile ]; then
  (echo -n 'someservice: ')\
    >/dev/console
  /etc/someservice
  (echo 'ok') >/dev/console
fi
```

This would have the effect of displaying "someservice: " (with no newline) just before starting the service, and finishing the line with "ok" once the service had started.

We also changed things so that each service printed an entire line of text (using `echo` without the `-n` option), rather than stringing messages from several services together into one line of text. It is

trivial to apply these changes to the more common `echo -n` style of messages.

Add Verbose Mode

For various reasons, RC files are often set up so that the services do not have stderr and stdout directed at `/dev/console` (typically so that they do not acquire a controlling terminal). This means that various warnings and error messages might easily go unnoticed by the system administrator who is trying to find out why the automount daemon is not being started even though it is in the RC file (for example). So we added code similar to the following near the start of each of our RC files (this example is from `/etc/rc`):

```
#
# $- is a list of the sh options that
# are set.  If -x isn't set, do the
# verbose thing.
#
case $- in
  *x*)
    ;;
  *)
    if [ -r /verbose ]; then
      exec sh -x /etc/rc \
        >/dev/console 2>&1
    fi
    ;;
esac
```

If a file named `/verbose` exists, then this part of the script will reexecute `/etc/rc` with `/bin/sh -x` and with stdout and stderr redirected to `/dev/console`. The `-x` option to `sh` causes it to echo each command to stdout before executing it. Each RC file starts with similar chunk of code, with the name of the script changed of course. At the end of `/etc/rc` we delete `/verbose` if it exists so that subsequent reboots are not done verbosely.

To reboot a host in "verbose" mode, you simply create the `/verbose` file and reboot the host. As it boots you will see all of the command lines as they are executed, and any messages that they print to stdout or `stderr`.

Reclaim rc.local for Local Use

Most vendors have added their own startup code to the `/etc/rc.local` file, which means that there is no place to put "real" local modifications and keep them distinct and separate from vendor supplied files. So we rearranged our RC files by pushing all of the vendor supplied code from `/etc/rc.local` back into `/etc/rc`, thereby reserving `/etc/rc.local` for our own use. This was aesthetically and philosophically pleasing, and made it easier to adapt our RC files on many hosts, though I suppose it has not really saved us any grief.

One Set of RC File for Many Hosts

We have 23 Sun file servers that are all essentially identical, and 260+ Sun workstations that are even more identical. We wanted to minimize the differences between the RC files on these hosts so that we could make the process of changing, testing and installing changes to the RC files as simple and convenient as possible. We accomplished this in part by adding conditions to the `rc.local` file that use the hostname to determine what services to start on which hosts. We also removed all of the other host specific information from the RC files and put them in a file that we call `/etc/rc.config`.

Here is a sample `rc.config` file from our site:

```
# Configuration information.
# Set values to "" when they
# don't apply to your setup.

# Network interfaces
net0=le0
net1=

# Hostnames
net0_name=brachiosaur
net1_name=

# Internet addresses
net0_address=128.146.12.10
net1_address=

# Netmasks...leave blank if
# you want to use the 'default'
# netmask, or set it to '+'
# to have ifconfig read
# /etc/netmasks.
net0_netmask=255.255.255.0
net1_netmask=

# Broadcast addresses, leave
# blank for defaults
net0_broadcast=128.146.12.255
net1_broadcast=

# YP domain name
ypdomain='cis.osu.new'

# Where to get the date from
rdate_from=dinosaur

# Default gateway
default_gw=dinosaur
```

At the start of each of the other RC files, we include the following code to read in `/etc/rc.config`:

```
# Read the host config info file.

. /etc/rc.config
case $? in
  0)
    ;;
  *)
    (echo "An error occurred while\
reading /etc/rc.config.") >/dev/console
    exit 1
    ;;
esac
```

The other RC scripts are set up to use these variables to control their actions (for example, the host name is set from net0_name).

The combination of the conditions in `/etc/rc.local` and the isolation of other host-specific information to `/etc/rc.config` allows us to use the same `rc.boot`, `rc.single`, `rc` and `rc.local` files on all of our Suns.

Conclusions

These changes have collectively made our lives a little easier here. The most useful changes were to switch the order of the echo messages with its associated command, and to add the /verbose mode. Reclaiming `/etc/rc.local` was aesthetically pleasing, and made it easier to isolate local additions from vendor additions. Using the `rc.config` file for the remaining differences has made it very easy to write automatic "builder" scripts that can generate client and server configurations from a standard prototype distribution.

The only real problem with our changes is that we have to carefully check our RC files against the vendor supplied files when we install new vendor releases so that we can pick up any changes or additions. Since we only have to do that once for each release and can readily distribute the modified RC files to all of our Suns, this has not proven to be overly burdensome.

One should be able to readily apply these changes to the RC files of any system that uses Bourne or C-shell scripts for startup configuration.

Author Information

Steve Romig is the software staff manager for the CIS Department at The Ohio State University. He received a B.S. degree in applied math from Carnegie Mellon University in 1982 and is slowly working toward an M.S. degree in computer science at Ohio State. His main professional interests are in simplifying and automating system administration tasks and in computer security. His email address is romig@cis.ohio-state.edu, his U.S. Mail address is The Ohio State University; Department of Computer and Information Science; 2036 Neil Avenue Mall; Columbus, OH 43210, and his telephone number is 614-292-8018.

Host Aliases and Symbolic Links -or- How to Hide the Servers' Real Name

John F. Detke - Octel Communications Corporation

ABSTRACT

This paper describes HASL, a methodology used to maintain resource locations across a heterogeneous network. HASL uses host aliases instead of primary host names for network references such as mounts and PC-NFS print redirection, and standardizes export paths. While resources move from host to host, or from controller to controller, the client's reference remains constant. HASL eliminates the need to update client network files, and allows a single coherent resource location map to be maintained across the network, independent of actual resource location.

Introduction

Maintaining PC-NFS hosts on our network used to require extensive effort, as few of the standard UNIX tools are available, and most PC-NFS users were not able to maintain their own machine. Since PC-NFS is a client-only architecture, administrative changes needed to be made from each PC's console, resulting in an extremely mobile and tired administrator. Maintaining SunOS clients was somewhat easier since most of the tools are present and changes can be made over the network. It was still necessary to update each client's configuration when network resources moved. Moving a widely mounted filesystem was a major task, involving changes to many clients and trips to many cubicles spread over multiple buildings. For reasons beyond our control we run SunOS 4.0.3, and so don't feel we could make use of Sun's automounter, and were too busy updating PC network files to install adm, and even if we could run adm on the SunOS clients, we would still have to visit each PC to update configuration files. While replacing edlin with a vi-like editor made this task less painful, mistakes were still made as PC-NFS network files are not confined to a single area but can be located almost anywhere.

Since I couldn't remotely update PC-NFS nodes, and really didn't want to edit any client configuration files, I attempted to design a system that would work with existing tools and eliminate the need for altering client configuration files. I could alter the network maps (hosts, ethers, etc) since all machines subscribe to NIS, and remount NFS filesystems. Even the PC-NFS users could remount their files, usually via the 3 fingered salute, aka rebooting the PC. I named the ensuing system HASL: "Host Aliasing and Symbolic Links", since it used both. HASL is a methodology, not a new tool. A name for this methodology was needed, so that I could talk and write this paper about it and distinguish between clients using HASL and those not. Naming HASL eliminated conversations along the lines of "Has that client been updated to use the new method of referring to servers by an alias?".

HASL: The Solution

HASL is a method for using existing tools. It involves using the aliasing feature of IP host addresses, and standardizing export paths for filesystems.

Aliasing Server Names

Each server that exports a filesystem, or supports PC printing is known by at least two names, the primary hostname and an alias associated with the network resource. Servers with multiple exported filesystems, or multiple printers, have multiple aliases. References to exported filesystems and print services occur only through the alias. Primary hostnames remain constant, while aliases follow the respective resource as it moves around the network. Since clients access a resource through an alias, client configuration files remain unchanged.

An example best illustrates this point. Say we have a filesystem "eng1", two servers valadimir and ludwig, and a client johann. The resource alias would be something like "eng1_serv". Currently the eng1 filesystem is located on valadimir, so that the host entries look like:

```
192.9.200.10   valadimir eng1_serv
192.9.200.20   ludwig
192.9.200.30   johann
```

and johann's fstab contains the entry:

```
eng1_serv:/Exports/eng1 /home/eng1 nfs rw
```

More on export and mount paths later. Now imagine that eng1 is moved from valadimir to ludwig. The host map would be modified to read:

```
192.9.200.10     valadimir
192.9.200.20     ludwig eng1_serv
192.9.200.30     johann
```

That is, the alias now points to the server ludwig.

Following the move, the client johann need only remount eng1, his fstab entry remains the same.

Remounting of moved filesystems was necessary before HASL, but often resulted in errors due to mistakes in the fstb or exports entries. These errors no longer occur, as the entries don't change. Users and operators no longer care where the eng1 filesystem is located. They only need to remember eng1 is mountable from the host eng1_serv, and to remount if eng1 ever moves.

Resources are advertised by the alias name and the export path. Such information is valid for the resource's life, and works equally well for PC-NFS clients' "mounting" of printers. A typical printer mount command looks something like:

```
net use lpt1:   lj1_serv lj1
```

which would allow the PC to print to lj1 via the printer named lpt1. If lj1 moves, the alias changes, and the net use command remains the same.

Standardized Export Paths.

HASL involves standardizing export paths, such that all servers export their filesystems in a consistent manner, and all a server's partitions are exported from a single directory. The directory /Exports is found on every server, and all files are exported from there. Within /Exports are symbolic links to where the exported partitions are actually mounted on the server. Thus, server mount points may change without affecting exports entries or clients' mount path.

Much like aliases, symbolic links are used to hide actual locations, and provide a consistent export path throughout the resource's life. The server's /etc/exports file and a client's fstab file remain relatively constant, simplifying partition movement and the information operators are required to remember and act upon.

When a resource moves to a new mount point on the same server, only the link is altered, the /etc/exports entry remains the same. If a resource is moved to another server, the identical /etc/exports entry is moved along with it. A client's reference to a resource may remain unchanged throughout the resource's life.

Again examples best illustrate the point. I will explain moving the filesystem "eng1" on server valadimir from /dev/sd1g to /dev/sd2g, and then moving eng1 to another server named frederic.

Initially eng1 is located on /dev/sd1g, which is mounted on valadimir:/Disks/eng1, and exported from /Exports/eng1 so that valadimir's fstab contains:

```
/dev/sd1g /Disks/sd1g 4.2 rw 1 2
```

and the /etc/exports entry reads

```
/Exports/eng1   -access=nfsclients
```

/Exports/eng1 is a symbolic link pointing to /Disks/sd1g.

The move is accomplished by

- copying the contents of /dev/sd0c to /dev/sd1g
- Relinking /Exports/eng1 to point to /Disks/sd1g
- frederic and the other clients remount eng1

Note frederic still references eng1 as: "eng1_serv:/Exports/eng1", and client configuration files, and valadimir's /etc/exports remain unchanged.

The eng1 partition may be moved to frederic without involving any changes on the client side. The steps would be:

- Update the eng1_server alias to reference frederic instead of the old server.
- Mount /Disks/sd1g on frederic
- Create the link /Exports/eng1, pointing to this mount.
- Add the entry for the link just created in frederic's exports file. The entry may look something like "/Exports/eng1 -access=nfsclients".
- Run exportfs to actually export this file system from frederic.
- Mount the new location on the clients.

Since no client configuration files are altered, there is no danger of spelling mistakes, nor the resultant help desk calls.

Installation

Since HASL is a methodology, rather than an actual program, it isn't installed as such, but more adopted. Adopting HASL involves adding aliases, standardizing export paths and changing client configuration files, hopefully for the last time. This conversion can be undertaken gradually, as the old methodology need not change immediately. The server aliases can be added without affecting anything, and the old export path may be retained while converting clients to reference the standardized export path. Once HASL is fully adopted, the old export paths can be deleted.

At Octel, we switched to HASL over a period of about 6 months. The aliases for the existing resources were added to the NIS database first, then for each resource we:

- Updated the server's export path, leaving the old path in place temporarily.
- Updated client's configuration files.

All Clients were not be updated simultaneously, changes were made when it was convenient. This gradual phasing in, one resource at a time, prevented catastrophic network changes. Leaving the old export paths intact meant that non-HASL clients were not disturbed. Once all clients were using HASL for a given resource, the old export path was removed. Occasionally this uncovered clients that

hadn't been updated, usually since a user missed or ignored the various notifications and instructions to convert. During this process, new resources were added. These resources were simply created using HASL, eliminating any need for conversion.

Experiences

We have had a few minor problems, mostly with the PC-NFS nodes. Some users don't remount when resources are moved, even though they are notified. When users attempt to access a moved resource without remounting, the attempt will fail, and they call the help desk. The users are happy with the response "that was moved, remount the resources" (which they usually interpreted as "reboot my machine"), more than "That was moved; alter your mount path to point to new_serv:/usr/local and remount".

DOS limitations lead to another problem: occasionally the DOS environment can't handle the large NIS responses generated when a single machine has multiple aliases. This may happen when a server exports multiple resources, so that multiple aliases exist for that server. For example the server "lucifer" may export 15 file systems, and so there would be 15 aliases for lucifer. We currently solve this problem several ways:

- Increasing the environment size.
- Decreasing the number of extraneous TSR's running.
- Associating one alias with multiple resources.

Multiple resources with a single alias somewhat defeats the purpose of HASL. Consequently, this tactic was used only as a last resort, on fairly static resources and with careful grouping. We have resorted to this technique only once, as a temporary measure while we diagnosed some unrelated PC-NFS/DOS problems. We haven't encountered any similar problems with UNIX machines, and generated some long aliases, and a large number of aliases during testing.

Since the number of mounts is limited in PC-NFS, not all resources can be mounted simultaneously, and so PC-NFS users have to know which resources they are interested in. If a user's home directory is moved, e.g. from eng1 to eng2, the user needs to change their configuration files, or notify the help desk so that we can. When we move a user's home directory or add resources, we provide plenty of advance notice, including explicit instructions for PC-NFS users on how to mount the new resource. The users seem to better understand mounting eng2 from eng2_serv than mounting eng2 from some arbitrary server such as "lucifer".

The difference between relative and effective working directory occasionally causes confusion. For example, the NIS might advertise joe's home directory as /Octel/eng1/joe, but some utilities may report ~joe as being /disks/eng1. This has not been a large problem at Octel so far. The shell seems to usually report the "correct" path.

HASL requires NFS filesystem starting out with a symbolic link, which may have performance implications. We would like to investigate this more, but we feel that the benefits of using the HASL methodology, standardized configuration files, fewer help calls, and less chance for human error, balance a minor performance penalty. We have not experienced large NFS performance problems using HASL.

One interesting side effect from not having to update each and every PC's files is that we no longer remember where all the PC's are located. This has necessitated adding cubicle numbers to the network maps so that we can locate a particular PC if the need arises. Currently this information is kept in the comment area in the ethers map.

Work In Progress

We are planning to run the automounter (when we can upgrade to SunOS 4.1.1) and haven't fully investigated how HASL would integrate. We don't expect many problems with running the automounter and HASL, and in fact they should get along quite well. Since the automounter should pick up any moves automatically, operators won't have to remount when resources move. There might be a problem with "active" mounts suddenly pointing to an empty location, which we plan to test for during beta implementations. The automounter should replace much of HASL's functions on the SunOS clients, but HASL will still be needed for the PC-NFS clients.

A DNS based network database is also in our future and may require some changes. We have looked into DNS, and since it supports aliasing, I don't expect many problems. I do use the NIS "feature" of including comments when ypcat'ing NIS files. I keep information such as cubicle number, machine serial number, OS level, and user in the comments, and expect to see that information when I ypcat a NIS file. I don't know of a DNS equivalent, but maybe we'll have a real database running by then.

Conclusions

HASL has been used at Octel Communications Corp., and has significantly reduced the administration load of PC-NFS nodes and SunOS clients, consequently reducing the number of help calls, and the help desk workload. Since we currently have two people staffing the help desk, and acting as operators, this has helped tremendously. The average PC-NFS user can generally handle their network configuration when provided with resource name, associated alias and the export path. When a resource moves, users/operators normally reboot, and everyone is happy.

Author Information

John F. Detke began sysadmining the usual way, because no one else wanted to, and his manager thought he had a little spare time. Every since that first network he has been attempting to mediate between the UNIX and DOS camps, at least somewhat successfully. In his spare time John enjoys discussing the merits of using Sun's NIS, and drawing maps of the Octel network. He can be Reached via U.S. Mail at Octel Communication Corp; M/S 05-04; 870 Tasman Dr; Milpitas, Ca 95035-7349 or via email at jfd@octel.com.

References

SunOS 4.1 Manual set

Redundant Printer Configuration

Steven C. Simmons - Industrial Technology Institute

ABSTRACT

As laser printers become smaller and cheaper, many sites are installing multiple conveniently located slow printers rather than one or two central fast printers. With an increase in numbers of printers and hosts supporting printers comes an increase in failures. This paper describes a naming and **/etc/printcap** convention for BSD-based spooling systems that makes it simple to relocate printers and redirect print jobs when failures or changes occur. As a side benefit, it makes **/etc/printcap** files and queues easier to manage.

Introduction

As the price of laser printers drops towards $1,000.00, many sites are buying a large number of cheap printers. This has many advantages: it lets printers be located conveniently, it provides redundant hardware on a network, etc, etc. Since these printers are usually located in areas close to those doing the printing, they are often attached to individual workstations or other small hosts.

When a printer or its supporting host fails, it is difficult to reconfigure the printers and network. The **/etc/printcap** files on all systems must usually be modified, and all affected queues halted and restarted. Even with automated tools this is a cumbersome and error-prone task.

We have developed some simple conventions which greatly ease the task of printer queue management. This technique works on all BSD-based spooling systems we have encountered (BSD, Ultrix, Gould/Encore, Sun) and may be useful in System V environments as well. The technique requires an easy way of modifying the host name database quickly, but may be useful even if such a feature is lacking.

The Conventions

The first requirement is to decouple the host name for the printer from any actual host name. Instead, host aliases and/or **CNAME** records should be installed so that every printer name (lw1, lw2, etc) is assigned a unique host name (lw1_host, lw2_host, etc).

Now a standard **/etc/printcap** file is created and distributed to all systems on the network. The file contains two entries for each printer, one defining a remote queue, and defining a local printer.

Remote entries for each printer should appear first. Each remote entry should point to a queue of the same name. The remote host alias should be used, not the true host name.

Local entries for each printer should follow the remote entries. The local entries should be commented out. (Note that in some versions of the BSD printer system it is not necessary to comment out extra entries with the same name; any entry after the first is ignored.)

On all hosts which actually have printers attached, the remote entries for those printers should be commented out and the matching local entries uncommented.

A simplified **/etc/printcap** example is shown below. We assume only two printers.

```
#
#  Standard Printcap File.
#
#  Remote printers
#
lw1|LaserWriter 1|:\
    :lp=:rm=lw1_host:rp=lw1:\
    :sd=/usr/spool/lp/lw1:\
    :lf=/usr/spool/lp/lw1/log:sh:
lw2|LaserWriter 2|:\
    :lp=:rm=lw2_host:rp=lw2:\
    :sd=/usr/spool/lp/lw2:\
    :lf=/usr/spool/lp/lw2/log:sh:
#
#  Local printers
#
#lw1|LaserWriter 1|:\
#    :sd=/var/spool/lp/lw1:\
#    :px#2400:py#2400:\
#    ...etc...
#lw2|LaserWriter 2|:\
#    :sd=/var/spool/lp/lw2:\
#    ...etc...
```

Basic Usage

Up to three actions are required when moving/repairing a printer. The hosts database must be modified to reflect the new or fill-in host for a given printer; the **/etc/printcap** on the old printer host may need to be modified; and the **/etc/printcap** on the new printer host will need to be modified.

Moving A Printer

When a printer moves from one host to another, the admin first stops and disables the queues on the old and new hosts. The host database should then be modified as needed, moving the alias or **CNAME** from the old printer host to the new.

On the host where the printer used to be attached, the admin comments out the local entry for the printer and uncomments the remote entry. On the host where the printer is now attached, the admin uncomments the local entry for the printer and comments out the remote entry.

The queues are then restarted and reenabled on both systems. No actions need to be taken on any other systems.

Redirecting From A Dead Printer

If a printer is out of service for a short time, one can redirect jobs by modifying only **/etc/printcap** on the host which originally was home for the printer. If we assume lw1 is down, the **/etc/printcap** entry for lw1 on lw1_home should be modified from

```
lw1|LaserWriter 1|:\
    :lp=:rm=lw1_host:rp=lw1:\
    :sd=/usr/spool/lp/lw1:\
    :lf=/usr/spool/lp/lw1/log:sh:
```

to

```
lw1|LaserWriter 1|:\
    :lp=:rm=lw2_host:rp=lw2:\
    :sd=/usr/spool/lp/lw1:\
    :lf=/usr/spool/lp/lw1/log:sh:
```

Note that only the remote host name and remote queue name are modified. All jobs intended for lw1 will still route to lw1_host, but will then be forwarded by it to lw2_host for the lw2 printer.

This is an increase in network overhead, as each job intended for lw1 is handled twice, first by lw1_host, and then by lw2_host. For short term outages, this is an acceptable overhead.

Recovering Queued Jobs

Once the queue has been redirected, simply halting and restarting the queue on the old lw1_host will cause all pending jobs for lw1 to be redirected to lw2.

Queue Management

One unexpected benefit of this system was that for many tasks the admins no longer needed to know the host name of the host supporting the printer. Simply using the alias for *rsh*, *rlogin* or *ping* served just as well. Most queue status changes could be handled by *rsh lwX_host lpc <action>*.

Determining which host actually represented the printer was a simple as a *grep* of the **/etc/hosts** or using *nslookup* or *ypmatch*.

Some Cautionary Notes

As in any system, there can be abuse. Do not let yourself fall into the trap of chaining jobs from one print queue to another to another. This will eventually cause a problem, and you'll forget where all the changes were made.

Beware of circular references. Some spooler systems will merrily let you do

```
lw1|LaserWriter 1|:\
    :lp=:rm=lw1_host:rp=lw1:\
    :sd=/usr/spool/lp/lw1:\
    :lf=/usr/spool/lp/lw1/log:sh:
```

on lw1_host, resulting in lots of CPU usage and very little printing.

Future Work

This same method should be quite useful in other circumstances. In complex networks where static routing is desired host aliases such as *net1_gateway* and *default_route* would greatly ease network changes. Host names like *news_host* and *yp_master* have obvious utility.

Conclusion

We have used this facility at several installations where there were multiple printers of the same types spread across a number of hosts. It has both made administration easier and things more convenient for the users.

Author Information

Steve Simmons is the System Support Manager at the Industrial Technology Institute and a consultant on system and network administration. He received his bachelors degree from the University of Michigan in 1980. He may be reached at Industrial Technology Institute, P.O.Box 1485, Ann Arbor, MI, 48109, 1-313-769-4086, email scs@iti.org; or at Inland Sea, 9353 Hidden Lake Circle, Dexter, MI, 48130, 1-313-426-8981, scs@lokkur.dexter.mi.us.

The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710
Telephone: 510-528-8649
Email: office@usenix.org
Fax: 510-548-5738

USENIX Supporting Members

Aerospace Corporation
AT&T Information Systems
Digital Equipment Corporation
Frame Technology Corporation
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corp.
Sun Microsystems, Inc.
Sybase, Inc.
UUNET Technologies, Inc.